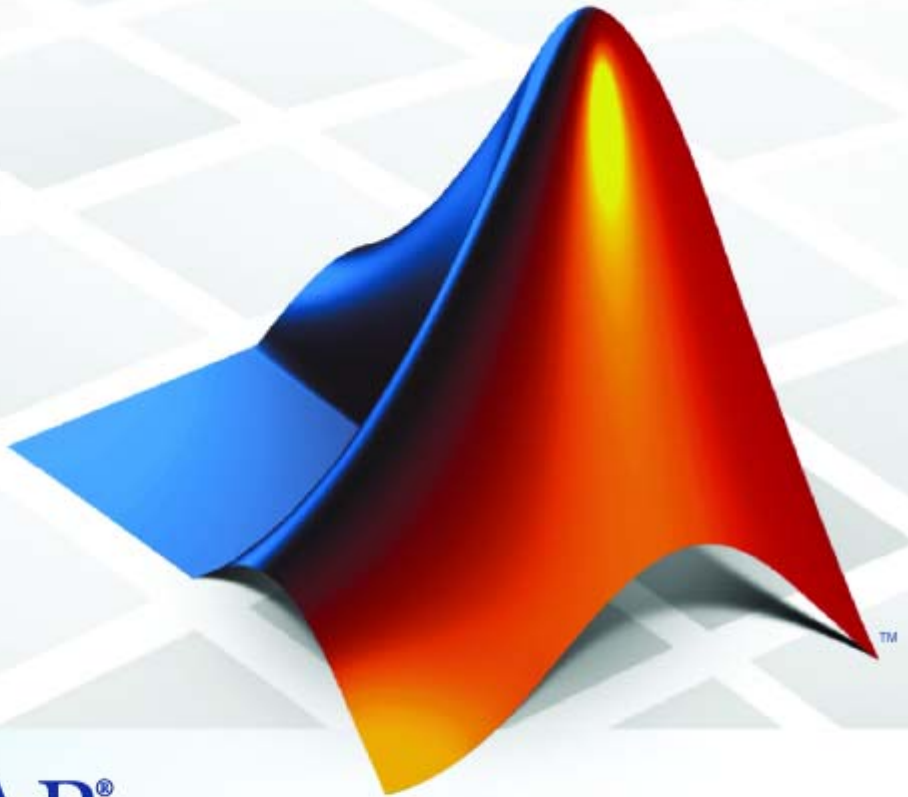


Embedded IDE Link™ 4

User's Guide

For Use with Texas Instruments' Code Composer Studio™



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Embedded IDE Link™ User's Guide

© COPYRIGHT 2002–2009 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

July 2002	Online only	New for Version 1.0 (Release 13)
October 2002	Online only	Revised for Version 1.1
May 2003	Online only	Revised for Version 1.2
September 2003	Online only	Revised for Version 1.3 (Release 13SP1+)
June 2004	Online only	Revised for Version 1.3.1 (Release 14)
October 2004	Online only	Revised for Version 1.3.2 (Release 14SP1)
December 2004	Online only	Revised for Version 1.4 (Release 14SP1+)
March 2005	Online only	Revised for Version 1.4.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.4.2 (Release 14SP3)
March 2006	Online only	Revised for Version 1.5 (Release 2006a)
April 2006	Online only	Revised for Version 2.0 (Release 2006a+)
September 2006	Online only	Revised for Version 2.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.0 (Release 2007a)
September 2007	Online only	Revised for Version 3.1 (Release 2007b)
March 2008	Online only	Revised for Version 3.2 (Release 2008a)
October 2008	Online only	Revised for Version 3.3 (Release 2008b)
March 2009	Online only	Revised for Version 3.4 (Release 2009a)
September 2009	Online only	Revised for Version 4.0 (Release 2009b)

Getting Started

1

Product Overview	1-2
Automation Interface	1-3
Project Generator	1-4
Verification	1-5
Product Features Supported for Each Processor Family ..	1-5
Configuration Information	1-6
Verifying Your Code Composer Studio Installation	1-6
Software Requirements	1-8

Automation Interface

2

Getting Started with Automation Interface	2-2
Introducing the Automation Interface Tutorial	2-2
Selecting Your Processor	2-6
Creating and Querying Objects for CCS IDE	2-8
Loading Files into CCS	2-10
Working with Projects and Data	2-12
Closing the Links or Cleaning Up CCS IDE	2-18
Getting Started with RTDX	2-20
Introducing the Tutorial for Using RTDX	2-21
Creating the tics Objects	2-26
Configuring Communications Channels	2-29
Running the Application	2-31
Closing the Connections and Channels or Cleaning Up ...	2-38
Listing Functions	2-41

Constructing tics Objects	2-42
Example — Constructor for tics Objects	2-42
tics Properties and Property Values	2-44
Overloaded Functions for tics Objects	2-45
tics Object Properties	2-46
Quick Reference to tics Object Properties	2-46
Details About tics Object Properties	2-48
Managing Custom Data Types with the Data Type	
Manager	2-54
Adding Custom Type Definitions to MATLAB	2-56

Project Generator

3

Introducing Project Generator	3-2
Project Generation and Board Selection	3-3
Schedulers and Timing	3-5
Configuring Models for Asynchronous Scheduling	3-5
Cases for Using Asynchronous Scheduling	3-6
Comparing Synchronous and Asynchronous Interrupt	
Processing	3-8
Using Synchronous Scheduling	3-10
Using Asynchronous Scheduling	3-10
Multitasking Scheduler Examples	3-11
Project Generator Tutorial	3-24
Creating the Model	3-25
Adding the Target Preferences Block to Your Model	3-25
Specify Configuration Parameters for Your Model	3-29

Setting Code Generation Parameters for TI Processors	3-33
Setting Model Configuration Parameters	3-36
Target File Selection	3-37
Build Process	3-38
Custom Storage Class	3-38
Report Options	3-38
Debug Pane Parameters	3-39
Optimization Pane Parameters	3-40
Embedded IDE Link Pane Parameters	3-42
Default Project Configuration — Custom	3-47
Using Custom Source Files in Generated Projects	3-48
Preparing to Replace Generated Files With Custom Files	3-48
Replacing Generated Source Files with Custom Files When You Generate Code	3-50
Optimizing Embedded Code with Target Function Libraries	3-52
About Target Function Libraries and Optimization	3-52
Using a Processor-Specific Target Function Library to Optimize Code	3-54
Process of Determining Optimization Effects Using Real-Time Profiling Capability	3-55
Reviewing Processor-Specific Target Function Library Changes in Generated Code	3-56
Reviewing Target Function Library Operators and Functions	3-58
Creating Your Own Target Function Library	3-58
Model Reference	3-59
How Model Reference Works	3-59
Using Model Reference	3-60
Configuring processors to Use Model Reference	3-62

4

What Is Verification?	4-2
Verifying Generated Code via Processor-in-the-Loop ..	4-3
What is Processor-in-the-Loop Cosimulation?	4-3
About the PIL Block	4-4
Preparing Your Model to Generate a PIL Application	4-5
Setting Model Configuration Parameters to Generate the PIL Application	4-6
Creating the PIL Block Application from a Model Subsystem	4-6
Running Your PIL Application to Perform Cosimulation and Verification	4-7
PIL Issues and Limitations	4-7
Profiling Code Execution in Real-Time	4-9
Overview	4-9
Profiling Execution by Tasks	4-10
Profiling Execution by Subsystems	4-12
System Stack Profiling	4-17
Overview	4-17
Profiling System Stack Use	4-19

Exporting Filter Coefficients from FDATool

5

About FDATool	5-2
Preparing to Export Filter Coefficients to Code	
Composer Studio Projects	5-4
Features of a Filter	5-4
Selecting the Export Mode	5-5
Choosing the Export Data Type	5-6

Exporting Filter Coefficients to Your Code Composer Studio Project	5-9
Exporting Filter Coefficients from FDATool to the CCS IDE Editor	5-9
Reviewing ANSI C Header File Contents	5-12
Preventing Memory Corruption When You Export Coefficients to Processor Memory	5-15
Allocating Sufficient or Extra Memory for Filter Coefficients	5-15
Example: Using the Exported Header File to Allocate Extra Processor Memory	5-15
Replacing Existing Coefficients in Memory with Updated Coefficients	5-16
Example: Changing Filter Coefficients Stored on Your Processor	5-17

Function Reference

6

Operations on Objects for CCS IDE	6-2
Operations on Objects for RTDX	6-4

Functions — Alphabetical List

7

Block Reference

8

Block Library: idelinklib_ticcs	8-2
Block Library: idelinklib_common	8-3

9

Configuration Parameters

10

Embedded IDE Link Pane	10-2
Overview	10-4
Export IDE link handle to base workspace	10-5
IDE link handle name	10-7
Profile real-time execution	10-8
Profile by	10-10
Number of profiling samples to collect	10-12
Project options	10-14
Compiler options string	10-16
Linker options string	10-18
System stack size (MAUs)	10-20
Build action	10-21
Interrupt overrun notification method	10-24
Interrupt overrun notification function	10-26
PIL block action	10-27
Maximum time allowed to build project (s)	10-29
Maximum time to complete IDE operations (s)	10-31
Source file replacement	10-33

Supported Processors

A

Supported Platforms	A-2
Product Features Supported by Each Processor or Family	A-2
Coemulation Support	A-3
Supported Processors and Simulators	A-3
Custom Board Support	A-4
 Supported Versions of Code Composer Studio	 A-5

Reported Limitations and Tips

B

Reported Issues	B-2
Demonstration Programs Do Not Run Properly Without Correct GEL Files	B-3
Error Accessing type Property of ticcs Object Having Size Greater Than 1	B-3
Changing Values of Local Variables Does Not Take Effect	B-4
Code Composer Studio Cannot Find a File After You Halt a Program	B-4
C54x XPC Register Can Be Modified Only Through the PC Register	B-6
Working with More Than One Installed Version of Code Composer Studio	B-6
Changing CCS Versions During a MATLAB Session	B-7
MATLAB Hangs When Code Composer Studio Cannot Find a Board	B-7
Using Mapped Drives	B-9
Uninstalling Code Composer Studio 3.3 Prevents Embedded IDE Link From Connecting	B-9

Index

Getting Started

- “Product Overview” on page 1-2
- “Configuration Information” on page 1-6
- “Software Requirements” on page 1-8

Product Overview

In this section...
“Automation Interface” on page 1-3
“Project Generator” on page 1-4
“Verification” on page 1-5
“Product Features Supported for Each Processor Family” on page 1-5

Embedded IDE Link™ software enables you to use MATLAB® functions to communicate with Code Composer Studio™ software and with information stored in memory and registers on a processor. With the `ticcs` objects, you can transfer information to and from Code Composer Studio software and with the embedded objects you get information about data and functions stored in your signal processor memory and registers, as well as information about functions in your project.

Embedded IDE Link lets you build, test, and verify automatically generated code using MATLAB, Simulink®, Real-Time Workshop®, and the Code Composer Studio integrated development environment. Embedded IDE Link makes it easy to verify code executing within the Code Composer Studio software environment using a model in Simulink software. This processor-in-the-loop testing environment uses code automatically generated from Simulink models by Real-Time Workshop® Embedded Coder™ software. A wide range of Texas Instruments DSPs are supported:

- TI's C2000™
- TI's C5000™
- TI's C6000™

With Embedded IDE Link , you can use MATLAB software and Simulink software to interactively analyze, profile and debug processor-specific code execution behavior within CCS. In this way, Embedded IDE Link automates deployment of the complete embedded software application and makes it easy for you to assess possible differences between the model simulation and processor code execution results.

Embedded IDE Link consists of these components:

- Project Generator—generate C code from Simulink models
- Automation Interface—use functions in the MATLAB command window to access and manipulate data and files in the IDE and on the processor
- Verification—verify how your programs run on your processor

With Embedded IDE Link, you create objects that connect MATLAB software to Code Composer Studio software. For information about using objects, refer to “Software Requirements” on page 1-8.

Note Embedded IDE Link uses objects. You work with them the way you use all MATLAB objects. You can set and get their properties, and use their methods to change them or manipulate them and the IDE to which they refer.

The next sections describe briefly the components of Embedded IDE Link software.

Automation Interface

The automation interface component is a collection of methods that use the Code Composer Studio API to communicate between MATLAB software and Code Composer Studio. With the interface, you can do the following:

- Automate complex tasks in the development environment by writing MATLAB software scripts to communicate with the IDE, or debug and analyze interactively in a live MATLAB software session.
- Automate debugging by executing commands from the powerful Code Composer Studio software command language.
- Exchange data between MATLAB software and the processor running in Code Composer Studio software.
- Set breakpoints, step through code, set parameters and retrieve profiling reports.
- Automate project creation, including adding source files, include paths, and preprocessor defines.

- Configure batch building of projects.
- Debug projects and code.
- Execute API Library commands.

The automation interface provides an application program interface (API) between MATLAB software and Code Composer Studio. Using the API, you can create new projects, open projects, transfer data to and from memory on the processor, add files to projects, and debug your code.

Project Generator

The Project Generator component is a collection of methods that use the Code Composer Studio API to create projects in Code Composer Studio and generate code with Real-Time Workshop. With the interface, you can do the following:

- Automated project-based build process
Automatically create and build projects for code generated by Real-Time Workshop or Real-Time Workshop Embedded Coder.
- Customize code generation
Use Embedded IDE Link with any Real-Time Workshop system target file (STF) to generate processor-specific and optimized code.
- Customize the build process
- Automate code download and debugging
Rapidly and effortlessly debug generated code in the Code Composer Studio software debugger, using either the instruction set simulator or real hardware.
- Create and build CCS projects from Simulink software models. Project Generator uses Real-Time Workshop software or Real-Time Workshop Embedded Coder software to build projects that work with C2000™ software, C5000™ software, and C6000™ software processors.
- Highly customized code generation with the system target file `ccslink_ert.tlc` and `ccslink_grt.tlc` that enable you to use the Configuration Parameters in your model to customize your generated code.

- Automate the process of building and downloading your code to the processor, and running the process on your hardware.

Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded IDE Link combine to provide the following verification tools for you to apply as you develop your code:

Processor in the Loop Cosimulation

Use cosimulation techniques to verify generated code running in an instruction set simulator or real processor environment.

Execution Profiling

Gather execution profiling timing measurements with Code Composer Studio to establish the timing requirements of your algorithm. See “Profiling Code Execution in Real-Time” on page 4-9.

Product Features Supported for Each Processor Family

Within the collection of processors that Embedded IDE Link supports, some subcomponents of the product do not apply. For the complete list of which features work with each processor or family, refer to “Product Features Supported by Each Processor or Family” on page A-2.

Configuration Information

To determine whether Embedded IDE Link is installed on your system, type this command at the MATLAB software prompt.

```
ver
```

When you enter this command, MATLAB software displays a list of the installed products. Look for a line similar to the following:

```
Embedded IDE Link          Version 4.x    (Release Specifier)
```

To get a bit more information about the software, such as the functions provided and where to find demos and help, enter the following command at the prompt:

```
help ticcs
```

If you do not see the listing, or MATLAB software does not recognize the command, you need to install Embedded IDE Link. Without the software, you cannot use MATLAB software with the objects to communicate with CCS.

Note For up-to-date information about system requirements, see “Software Requirements” on page 1-8.

Verifying Your Code Composer Studio Installation

To verify that CCS is installed on your machine and has at least one board configured, enter

```
ccsboardinfo
```

at the MATLAB software command line. With CCS installed and configured, MATLAB software returns information about the boards that CCS recognizes on your machine, in a form similar to the following listing.

```
Board Board          Proc Processor  Processor
Num  Name            Num  Name         Type
-----
```

```
1 C6xxx Simulator (Texas Instrum .0 6701 TMS320C6701
0 C6x13 DSK (Texas Instruments) 0 CPU TMS320C6x1x
```

If MATLAB software does not return information about any boards, open your CCS installation and use the Setup Utility in CCS to configure at least one board.

As a final test, start CCS to ensure that it starts up successfully. For Embedded IDE Link to operate with CCS, the CCS IDE must be able to run on its own.

Embedded IDE Link uses objects to create:

- Connections to the Code Composer Studio Integrated Development Environment (CCS IDE)
- Connections to the RTDX™ (RTDX) interface. This object is a subset of the object that refers to the CCS IDE.

Concepts to know about the objects in this toolbox are covered in these sections:

- Constructing Objects
- Properties and Property Values
- Overloaded Functions for Links

Refer to MATLAB Classes and Objects in your MATLAB documentation for more details on object-oriented programming in MATLAB software.

Many of the objects use COM server features to create handles for working with the objects. Refer to your MATLAB documentation for more information about COM as used by MATLAB software.

Software Requirements

For detailed information about the software and hardware required to use Embedded IDE Link software, refer to the Embedded IDE Link system requirements areas on the MathWorks Web site:

- Requirements for Embedded IDE Link:
www.mathworks.com/products/ide-link/requirements.html
- Requirements for use with Code Composer Studio:
www.mathworks.com/products/ide-link/ti-adaptor.html

Automation Interface

- “Getting Started with Automation Interface” on page 2-2
- “Getting Started with RTDX” on page 2-20
- “Constructing ticcs Objects” on page 2-42
- “ticcs Properties and Property Values” on page 2-44
- “Overloaded Functions for ticcs Objects” on page 2-45
- “ticcs Object Properties” on page 2-46
- “Managing Custom Data Types with the Data Type Manager” on page 2-54

Getting Started with Automation Interface

In this section...
“Introducing the Automation Interface Tutorial” on page 2-2
“Selecting Your Processor” on page 2-6
“Creating and Querying Objects for CCS IDE” on page 2-8
“Loading Files into CCS” on page 2-10
“Working with Projects and Data” on page 2-12
“Closing the Links or Cleaning Up CCS IDE” on page 2-18

Introducing the Automation Interface Tutorial

Embedded IDE Link provides a connection between MATLAB software and a processor in CCS. You can use objects to control and manipulate a signal processing application using the computational power of MATLAB software. This approach can help you debug and develop your application. Another possible use for automation is creating MATLAB scripts that verify and test algorithms that run in their final implementation on your production processor.

Before using the functions available with the objects, you must select a processor to be your processor because any object you create is specific to a designated processor and a designated instance of CCS IDE. Selecting a processor is necessary for multiprocessor boards or multiple board configurations of CCS.

When you have one board with a single processor, the object defaults to the existing processor. For the objects, the simulator counts as a board; if you have both a board and a simulator that CCS recognizes, you must specify the processor explicitly.

To get you started using objects for CCS IDE software, Embedded IDE Link includes a tutorial that introduces you to working with data and files. As you work through this tutorial, you perform the following tasks that step you through creating and using objects for CCS IDE:

- 1 Select your processor.
- 2 Create and query objects to CCS IDE.
- 3 Use MATLAB software to load files into CCS IDE.
- 4 Work with your CCS IDE project from MATLAB software.
- 5 Close the connections you opened to CCS IDE.

The tutorial provides a working process (a *workflow*) for using Embedded IDE Link and your signal processing programs to develop programs for a range of Texas Instruments™ processors.

During this tutorial, you load and run a digital signal processing application on a processor you select. The tutorial demonstrates both writing to memory and reading from memory in the “Working with Projects and Data” on page 2-12” portion of the tutorial.

You can use the `read` and `write` methods, as described in this tutorial, to read and write data to and from your processor.

The tutorial covers the object methods and functions for Embedded IDE Link. The functions listed in the first table apply to CCS IDE independent of the objects — you do not need an object to use these functions. The methods listed in the second and third table requires a `ticcs` object that you use in the method syntax:

Functions for Working With Embedded IDE Link

The following functions do not require a `ticcs` object as an input argument:

Function	Description
<code>ccsboardinfo</code>	Return information about the boards that CCS IDE recognizes as installed on your PC.
<code>ticcs</code>	Construct an object to communicate with CCS IDE. When you construct the object you specify the processor board and processor.

Methods for Working with ticcs Objects

The methods in the following table require a ticcs object as an input argument:

Method	Description
address	Return the address and page for an entry in the symbol table in CCS IDE.
display	Display the properties of an object to CCS IDE and RTDX.
halt	Terminate execution of a process running on the processor.
info	Return information about the processor or information about open RTDX channels.
isrtdxcapable	Test whether your processor supports RTDX communications.
isrunning	Test whether the processor is executing a process.
read	Retrieve data from memory on the processor.
restart	Restore the program counter (PC) to the entry point for the current program.
run	Execute the program loaded on the processor.
visible	Set whether CCS IDE window is visible on the desktop while CCS IDE is running.
write	Write data to memory on the processor.

Methods for Embedded Objects

The methods in the following table enable you to manipulate programs and memory with an embedded object:

Method	Description
<code>list</code>	Return various information listings from Code Composer Studio software.
<code>read</code>	Read the information at the location accessed by an object into MATLAB software as numeric values. Demonstrated with a numeric, string, structure, and enumerated objects.
<code>write</code>	Write to the location referenced by an object. Demonstrated with numeric, string, structure, and enumerated objects.

Running Code Composer Studio Software on Your Desktop – Visibility

When you create a `ticcs` object, Embedded IDE Link starts CCS in the background.

When CCS IDE is running in the background, it does not appear on your desktop, in your task bar, or on the **Applications** page in the Task Manager. It does appear as a process, `cc_app.exe`, on the **Processes** tab in Microsoft® Windows Task Manager.

You can make the CCS IDE visible with the function `visible`. The function `invisible` returns the status of the IDE—whether it is visible on your desktop. To close the IDE when it is not visible and MATLAB software is not running, use the **Processes** tab in Microsoft Windows Task Manager and look for `cc_app.exe`.

If a link to CCS IDE exists when you close CCS, the application does not close. Microsoft Windows software moves it to the background (it becomes invisible).

Only after you clear all links to CCS IDE, or close MATLAB software, does closing CCS IDE unload the application. You can see if CCS IDE is running in the background by checking in the Microsoft Windows Task Manager. When CCS IDE is running, the entry `cc_app.exe` appears in the **Image Name** list on the **Processes** tab.

When you close MATLAB software while CCS IDE is not visible, MATLAB software closes CCS if it started the IDE. This happens because the operating system treats CCS as a subprocess in MATLAB software when CCS is not visible. Having MATLAB software close the invisible IDE when you close MATLAB software prevents CCS from remaining open. You do not need to close it using Microsoft Windows Task Manager.

If CCS IDE is not visible when you open MATLAB software, closing MATLAB software leaves CCS IDE running in an invisible state. MATLAB software leaves CCS IDE in the visibility and operating state in which it finds it.

Running the Interactive Tutorial

You have the option of running this tutorial from the MATLAB software command line or entering the functions as described in the following tutorial sections.

To run the tutorial in MATLAB software, click `run ccstutorial`. This command opens the tutorial in an interactive mode where the tutorial program provides prompts and text descriptions to which you respond to move to the next portion of the lesson. The interactive tutorial covers the same information provided by the following tutorial sections. You can view the tutorial M-file used here by clicking `ccstutorial.m`.

Selecting Your Processor

Links for CCS IDE provides two tools for selecting a board and processor in multiprocessor configurations. One is a command line tool called `ccsboardinfo` which prints a list of the available boards and processors. So that you can use this function in a script, `ccsboardinfo` can return a MATLAB software structure that you use when you want your script to select a board without your help.

Note The board and processor you select is used throughout the tutorial.

- 1 To see a list of the boards and processors installed on your PC, enter the following command at the MATLAB software prompt:

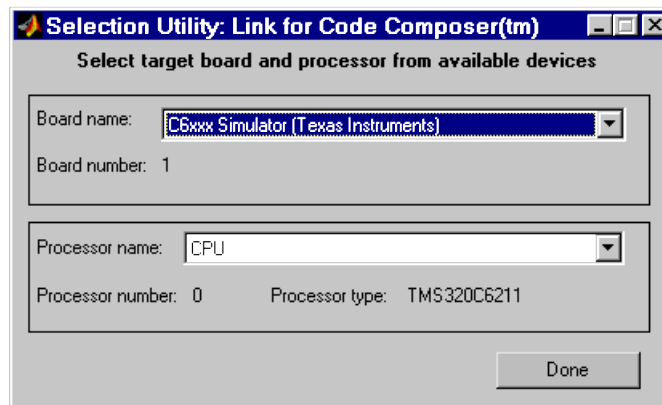
```
ccsboardinfo
```

MATLAB software returns a list that shows you all the boards and processors that CCS IDE recognizes as installed on your system.

- 2 To use the Selection Utility, `boardprocse1`, to select a board, enter

```
[boardnum,procnum] = boardprocse1
```

When you use `boardprocse1`, you see a dialog box similar to the following. Note that some entries vary depending on your board set.



- 3 Select a board name and processor name from the lists.

You are selecting a board and processor number that identifies your particular processor. When you create the object for CCS IDE in the next section of this tutorial, the selected board and processor become the processor of the object.

- 4 Click **Done** to accept your board and processor selection and close the dialog box.

boardnum and procnum now represent the **Board name** and **Processor name** you selected — boardnum = 1 and procnum = 0

Creating and Querying Objects for CCS IDE

In this tutorial section, you create the connection between MATLAB software and CCS IDE. This connection, or object, is a MATLAB software object that you save as variable `cc`.

You use function `ticcs` to create objects. When you create objects, `ticcs` input arguments let you define other object property values, such as the global timeout. Refer to the `ticcs` reference documentation for more information on these input arguments.

Use the generated object `cc` to direct actions to your processor. In the following tasks, `cc` appears in all function syntax that interact with CCS IDE and the processor:

- 1 Create an object that refers to your selected board and processor. Enter the following command at the prompt.

```
cc=ticcs('boardnum',boardnum,'procnum',procnum)
```

If you were to watch closely, and your machine is not too fast, you see Code Composer Studio software appear briefly when you call `ticcs`. If CCS IDE was not running before you created the new object, CCS starts and runs in the background.

- 2 Enter `visible(cc,1)` to force CCS IDE to be visible on your desktop.

Usually, you need to interact with Code Composer Studio software while you develop your application. The first function in this tutorial, `visible`, controls the state of CCS on your desktop. `visible` accepts Boolean inputs that make CCS either visible on your desktop (input to `visible` = 1) or invisible on your desktop (input to `visible` = 0). For this tutorial, use `visible` to set the CCS IDE visibility to 1.

- 3 Next, enter `display(cc)` at the prompt to see the status information.

```

TICCS Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

```

Embedded IDE Link provides three methods to read the status of a board and processor:

- `info` — Return a structure of testable board conditions.
- `display` — Print information about the processor.
- `isrunning` — Return the state (running or halted) of the processor.
- `isrtdxcapable` — Return whether the hardware supports RTDX.

4 Type `linkinfo = info(cc)`.

The `cc` link status information provides information about the hardware as follows:

```

linkinfo =
  boardname: 'C6711 Device Simulator'
  procname: 'CPU_1'
  isbigendian: 0
    family: 320
    subfamily: 103
    revfamily: 11
  procsortype: 'simulator'
  revsilicon: 0
    timeout: 10

```

5 Check whether the processor is running by entering

```
runstatus = isrunning(cc)
```

MATLAB software responds, indicating that the processor is stopped, as follows:

```
runstatus =  
  
0
```

- 6** At last, verify that the processor supports RTDX communications by entering

```
usesrtdx = isrtdxcapable(cc)  
usesrtdx =  
  
1
```

Loading Files into CCS

You have established the connection to a processor and board. Using three methods you learned about the hardware, whether it was running, its type, and whether CCS IDE was visible. Next, the processor needs something to do.

In this part of the tutorial, you load the executable code for the processor CPU in CCS IDE. Embedded IDE Link includes a CCS project file. Through the next tasks in the tutorial, you locate the tutorial project file and load it into CCS IDE. The open method directs CCS to load a project file or workspace file.

Note CCS has workspace and workspace files that are different from the MATLAB workspace files and workspace. Remember to monitor both workspaces.

After you have executable code running on your processor, you can exchange data blocks with it. Exchanging data is the purpose of the objects provided by Embedded IDE Link software.

- 1** To load the appropriate project file to your processor, enter the following command at the MATLAB software prompt. `getdemo`project is a specialized function for loading Embedded IDE Link demo files. It is not supported as a standard Embedded IDE Link function.

```

demoPjt= getDemoProject(cc,'ccstutorial')

demoPjt.ProjectFile

ans =

C:\Temp\LinkForCCSDemos_v3.2\ccstutorial\c6x\c67x\ccstut.pjt

demoPjt.DemoDir

ans =

C:\Temp\LinkForCCSDemos_v3.2\ccstutorial\c6x\c67x

```

Your paths may be different if you use a different processor. Note where the software stored the demo files on your machine. In general, Embedded IDE Link software stores the demo project files in

```
LinkforCCS_vproduct_version
```

Embedded IDE Link creates this directory in a location where you have write permission. There are two locations where Embedded IDE Link software tries to create the demo directory, in the following order:

- a** In a temporary directory on your C drive, such as `C:\temp\`.
 - b** If Embedded IDE Link software cannot use the `temp` directory, you see a dialog box that asks you to select a location to store the demos.
- 2** Enter the following command at the MATLAB command prompt to build the processor executable file in CCS IDE.

```
build(cc, 'all', 20)
```

You may get an error related to one or more missing `.lib` files. If you installed CCS IDE in a directory other than the default installation directory, browse in your installation directory to find the missing file or files. Refer to the path in the error message as an indicator of where to find the missing files.

- 3** Enter `load(cc, 'projectname.out')` to load the processor execution file, where *projectname* is the tutorial you chose, such as `ccstut_67x`.

You have a loaded program file and associated symbol table to the IDE and processor.

- 4** To determine the memory address of the global symbol `ddat`, enter the following command at the prompt:

```
ddata = address(cc, 'ddat')  
ddata =
```

```
1.0e+009 *  
2.1475      0
```

Your values for `ddata` may be different depending on your processor.

Note The symbol table is available after you load the program file into the processor, not after you build a program file.

- 5** To convert `ddata` to a hexadecimal string that contains the memory address and memory page, enter the following command at the prompt:

```
dec2hex(ddata)
```

MATLAB software displays the following response, where the memory page is `0x00000000` and the address is `0x80000010`.

```
ans =  
80000010  
00000000
```

Working with Projects and Data

After you load the processor code, you can use Embedded IDE Link functions to examine and modify data values in the processor.

When you look at the source file listing in the CCS IDE Project view window, there should be a file named `ccstut.c`. Embedded IDE Link ships this file with the tutorial and includes it in the project.

`ccstut.c` has two global data arrays — `ddat` and `idat` — that you declare and initialize in the source code. You use the functions `read` and `write` to access these processor memory arrays from MATLAB software.

Embedded IDE Link provides three functions to control processor execution — `run`, `halt`, and `restart`.

- 1 To demonstrate these commands, use the following function to add a breakpoint to line 64 of `cctut.c`.

```
insert(cc, 'cctut.c', 64)
```

Line 64 is

```
printf("Embedded IDE Link: Tutorial - Memory Modified by Matlab!\n");
```

For information about adding breakpoints to a file, refer to `insert` in the online Help system. Then proceed with the tutorial.

- 2 To demonstrate the new functions, try the following functions.

```
halt(cc)                % Halt the processor.
restart(cc)             % Reset the PC to start of program.
run(cc, 'runtohalt', 30); % Wait for program execution to stop at
                        % breakpoint (timeout = 30 seconds).
```

When you switch to viewing CCS IDE, you see that your program stopped at the breakpoint you inserted on line 64, and the program printed the following messages in the CCS IDE **Stdout** tab. Nothing prints in the MATLAB command window:

```
Embedded IDE Link: Tutorial - Initialized Memory
Double Data array = 16.3 -2.13 5.1 11.8
Integer Data array = -1-508-647-7000 (call me anytime!)
```

- 3 Before you restart your program (currently stopped at line 64), change some values in memory. Perform one of the following procedures based on your processor.

C5xxx processor family — Enter the following functions to demonstrate the `read` and `write` functions.

- a** Enter `ddatv = read(cc,address(cc,'ddat'),'double',4)`.

MATLAB software responds with

`ddatv =`

```
16.3000    -2.1300    5.1000    11.8000
```

- b** Enter `idatv = read(cc,address(cc,'idat'),'int16',4)`.

Now MATLAB software responds

`idatv =`

```
-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(cc,address(cc,'idat'),'int8',4)
```

`idatv =`

```
1 0 -4 1
```

- c** You can change the values stored in `ddat` by entering
`write(cc,address(cc,'ddat'),double([pi 12.3 exp(-1)...
sin(pi/4)]))`

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

- d** To change `idat`, enter

```
write(cc,address(cc,'idat'),int32([1:4]))
```

Here you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

- e** Start the program running again by entering the following command:

```
run(cc,'runtohalt',30);
```

The Stdout tab in CCS IDE reveals that `ddat` and `idat` contain new values. Next, read those new values back into MATLAB software.

- f** Enter `ddatv = read(cc,address(cc,'ddat'),'double',4)`.

```
ddatv =
```

```
3.1416 12.3000 0.3679 0.7071
```

`ddatv` contains the values you wrote in step c.

- g** Verify that the change to `idatv` occurred by entering the following command at the prompt:

```
idatv = read(cc,address(cc,'idat'),'int16',4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =
```

```
1 2 3 4
```

- h** Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(cc);
```

C6xxx processor family — Enter the following commands to demonstrate the `read` and `write` functions.

- a** Enter `ddatv = read(cc,address(cc,'ddat'),'double',4)`.

MATLAB software responds with

```
ddatv =
```

```
16.3000 -2.1300 5.1000 11.8000
```

- b** Enter `idatv = read(cc,address(cc,'idat'),'int16',4)`.

MATLAB software responds

```
idatv =
```

```
-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(cc,address(cc,'idat'),'int8',4)
```

```
idatv =
```

```
1 0 -4 1
```

- c** Change the values stored in `ddat` by entering

```
write(cc,address(cc,'ddat'),double([pi 12.3 exp(-1)...  
sin(pi/4)]))
```

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

- d** To change `idat`, enter the following command:

```
write(cc,address(cc,'idat'),int32([1:4]))
```

In this command, you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

- e** Next, start the program running again by entering the following command:

```
run(cc,'runtohalt',30);
```

The **Stdout** tab in CCS IDE reveals that `ddat` and `idat` contain new values. Read those new values back into MATLAB software.

- f** Enter `ddatv= read(cc,address(cc,'ddat'),'double',4)`.

```
ddatv =
```

```
3.1416 12.3000 0.3679 0.7071
```

Verify that `ddatv` contains the values you wrote in step c.

- g** Verify that the change to `idatv` occurred by entering the following command:

```
idatv = read(cc,address(cc,'idat'),'int32',4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =
1 2 3 4
```

- h** Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(cc);
```

- 4** Embedded IDE Link offers more functions for reading and writing data to your processor. These functions let you read and write data to the processor registers: `regread` and `regwrite`. They let you change variable values on the processor in real time. The functions behave slightly differently depending on your processor. Select one of the following procedures to demonstrate `regread` and `regwrite` for your processor.

C5xxx processor family — Most registers are memory-mapped and available using `read` and `write`. However, the PC register is not memory mapped. To access this register, use the special functions — `regread` and `regwrite`. The following commands demonstrate how to use these functions to read and write to the PC register.

- a** To read the value stored in register PC, enter the following command at the prompt to indicate to MATLAB software the data type to read. The input string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

```
cc.regread('PC','binary')
```

MATLAB software displays

```
ans =
33824
```

- b** To write a new value to the PC register, enter the following command. This time, the `binary` input argument tells MATLAB software to write the value to the processor as an unsigned binary integer. Notice that you used `hex2dec` to convert the hexadecimal string to decimal.

```
cc.regwrite('PC',hex2dec('100'),'binary')
```

- c Verify that the PC register contains the value you wrote.

```
cc.regread('PC','binary')
```

C6xxx processor family — `regread` and `regwrite` let you access the processor registers directly. Enter the following commands to get data into and out of the A0 and B2 registers on your processor.

- a To retrieve the value in register A0 and store it in a variable in your MATLAB workspace. Enter the following command:

```
treg = cc.regread('A0','2scomp');
```

`treg` contains the two's complement representation of the value in A0.

- b To retrieve the value in register B2 as an unsigned binary integer, enter the following command:

```
cc.regread('B2','binary');
```

- c Next, enter the following command to use `regwrite` to put the value in `treg` into register A2.

```
cc.regwrite('A2',treg,'2scomp');
```

CCS IDE reports that A0, B2, and A2 have the values you expect. Select **View > CPU Registers > Core Registers** from the CCS IDE menu bar to list the processor registers.

Closing the Links or Cleaning Up CCS IDE

Objects that you create in Embedded IDE Link software have COM handles to CCS. Until you delete these handles, the CCS process (`cc_app.exe` in the Microsoft Windows Task Manager) remains in memory. Closing MATLAB software removes these COM handles, but there may be times when you want to delete the handles without closing the application.

Use `clear` to remove objects from your MATLAB workspace and to delete handles they contain. `clear all` deletes everything in your workspace. To retain your MATLAB software data while deleting objects and handles, use `clear objname`. This applies both to `ticcs` objects you create with `ticcs` and other object you create with `createobj`. To remove the objects created

during the tutorial, the tutorial program executes the following command at the prompt:

```
clear cvar cfield uintcvar
```

This tutorial also closes the project in CCS with the following command:

```
close(cc,projfile,'project')
```

To delete your link to CCS, enter `clear cc` at the prompt.

Your development tutorial using Code Composer Studio IDE is done.

During the tutorial you

- 1** Selected your processor.
- 2** Created and queried links to CCS IDE to get information about the link and the processor.
- 3** Used MATLAB software to load files into CCS IDE, and used MATLAB software to run that file.
- 4** Worked with your CCS IDE project from MATLAB software by reading and writing data to your processor, and changing the data from MATLAB software.
- 5** Created and used the embedded objects to manipulate data in a C-like way.
- 6** Closed the links you opened to CCS IDE.

Getting Started with RTDX

In this section...
“Introducing the Tutorial for Using RTDX” on page 2-21
“Creating the ticcs Objects” on page 2-26
“Configuring Communications Channels” on page 2-29
“Running the Application” on page 2-31
“Closing the Connections and Channels or Cleaning Up” on page 2-38
“Listing Functions” on page 2-41

Support for using RTDX with C5000 and C6000 processors will be removed in a future release.

Embedded IDE Link and the objects for CCS IDE and RTDX speed and enhance your ability to develop and deploy digital signal processing systems on Texas Instruments processors. By using MATLAB software and Embedded IDE Link, your MathWorks™ tools, CCS IDE and RTDX work together to help you test and analyze your processing algorithms in your MATLAB workspace.

In contrast to CCS IDE, using links for RTDX lets you interact with your process in real time while it's running on the processor. Across the connection between MATLAB software and CCS, you can:

- Send and retrieve data from memory on the processor
- Change the operating characteristics of the program
- Make changes to algorithms as needed without stopping the program or setting breakpoints in the code

Enabling real-time interaction lets you more easily see your process or algorithm in action, the results as they develop, and the way the process runs.

This tutorial assumes you have Texas Instruments' Code Composer Studio™ software and at least one DSP development board. You can use the hardware simulator in CCS IDE to run this tutorial. The tutorial uses the TMS320C6711 DSK as the board, with the C6711 DSP as the processor.

After you complete the tutorial, either in the demonstration form or by entering the functions along with this text, you are ready to begin using RTDX with your applications and hardware.

Introducing the Tutorial for Using RTDX

Digital signal processing development efforts begin with an idea for processing data; an application area, such as audio or wireless communications or multimedia computing; and a platform or hardware to host the signal processing. Usually these processing efforts involve applying strategies like signal filtering, compression, and transformation to change data content; or isolate features in data; or transfer data from one form to another or one place to another.

Developers create algorithms they need to accomplish the desired result. After they have the algorithms, they use models and DSP processor development tools to test their algorithms, to determine whether the processing achieves the goal, and whether the processing works on the proposed platform.

Embedded IDE Link and the links for RTDX and CCS IDE ease the job of taking algorithms from the model realm to the real world of the processor on which the algorithm runs.

RTDX and links for CCS IDE provide a communications pathway to manipulate data and processing programs on your processor. RTDX offers real-time data exchange in two directions between MATLAB software and your processor process. Data you send to the processor has little effect on the running process and plotting the data you retrieve from the processor lets you see how your algorithms are performing in real time.

To introduce the techniques and tools available in Embedded IDE Link for using RTDX, the following procedures use many of the methods in the link software to configure the processor, open and enable channels, send data to the processor, and clean up after you finish your testing. Among the functions covered are:

Functions From Objects for CCS IDE

Function	Description
<code>ticcs</code>	Create connections to CCS IDE and RTDX.
<code>cd</code>	Change your CCS IDE working directory from MATLAB software.
<code>open</code>	Load program files in CCS IDE.
<code>run</code>	Run processes on the processor.

Functions From the RTDX Class

Function	Description
<code>close</code>	Close the RTDX links between MATLAB software and your processor.
<code>configure</code>	Determine how many channel buffers to use and set the size of each buffer.
<code>disable</code>	Disable the RTDX links before you close them.
<code>display</code>	Return the properties of an object in formatted layout. When you omit the closing semicolon on a function, <code>disp</code> (a built-in function) provides the default display for the results of the operation.
<code>enable</code>	Enable open channels so you can use them to send and retrieve data from your processor.
<code>isenabled</code>	Determine whether channels are enabled for RTDX communications.

Function	Description
isreadable	Determine whether MATLAB software can read the specified memory location.
iswritable	Determine whether MATLAB software can write to the processor.
msgcount	Determine how many messages are waiting in a channel queue.
open	Open channels in RTDX.
readmat	Read data matrices from the processor into MATLAB software as an array.
readmsg	Read one or more messages from a channel.
writemsg	Write messages to the processor over a channel.

This tutorial provides the following workflow to show you how to use many of the functions in the links. By performing the steps provided, you work through many of the operations yourself. The tutorial follows the general task flow for developing digital signal processing programs through testing with the links for RTDX.

Within this set of tasks, numbers 1, 2, and 4 are fundamental to all development projects. Whenever you work with MATLAB software and objects for RTDX, you perform the functions and tasks outlined and presented in this tutorial. The differences lie in Task 3. Task 3 is the most important for using Embedded IDE Link to develop your processing system.

- 1** Create an RTDX link to your desired processor and load the program to the processor.

All projects begin this way. Without the links you cannot load your executable to the processor.

- 2** Configure channels to communicate with the processor.

Creating the links in Task 1 did not open communications to the processor. With the links in place, you open as many channels as you need to support the data transfer for your development work. This task includes configuring channel buffers to hold data when the data rate from the processor exceeds the rate at which MATLAB software can capture the data.

- 3** Run your application on the processor. You use MATLAB software to investigate the results of your running process.
- 4** Close the links to the processor and clean up the links and associated debris left over from your work.

Closing channels and cleaning up the memory and links you created ensures that CCS IDE, RTDX, and Embedded IDE Link are ready for the next time you start development on a project.

This tutorial uses an executable program named `rtdxtutorial_6xevm.out` as your application. When you use the RTDX and CCS IDE links to develop your own applications, replace `rtdxtutorial_6xevm.out` in Task 3 with the filename and path to your digital signal processing application.

You can view the tutorial M-file used here by clicking `rtdxtutorial`. To run this tutorial in MATLAB software, click `run rtdxtutorial`.

Note To be able to open and enable channels over a link to RTDX, the program loaded on your processor must include functions or code that define the channels.

Your C source code might look something like this to create two channels, one to write and one to read.

```
    rtdx_CreateInputChannel(ichan); % processor reads from this.  
    rtdx_CreateOutputChannel(ochan); % processor writes to this.
```

These are the entries we use in `int16.c` (the source code that generates `rtdxtutorial_6xevm.out`) to create the read and write channels.

If you are working with a model in Simulink software and using code generation, use the To Rtdx and From Rtdx blocks in your model to add the RTDX communications channels to your model and to the executable code on your processor.

One more note about this tutorial. Throughout the code we use both the dot notation (direct property referencing) to access functions and link properties and the function form.

For example, use the following command to open and configure `ichan` for write mode.

```
cc.rtdx.open('ichan','w');
```

You could use an equivalent syntax, the function form, that does not use direct property referencing.

```
open(cc.rtdx,'ichan','w');
```

Or, use

```
open(rx,'ichan','w');
```

if you created an alias `rx` to the RTDX portion of `cc`, as shown by the following command:

```
rx = cc.rtdx;
```

Creating the tics Objects

With your processing model converted to an executable suitable for your desired processor, you are ready to use the objects to test and run your model on your processor. Embedded IDE Link and the objects do not distinguish the source of the executable — whether you used Embedded IDE Link and Real-Time Workshop, CCS IDE, or some other development tool to program and compile your model to an executable does not affect the object connections. So long as your `..out` file is acceptable to the processor you select, Embedded IDE Link provides the connection to the processor.

Note Program `rtdxtutorial_6xevm.out` uses the C6711. The executable is compiled, built, and linked to run on the C6711 processor. To use the tutorial without changes, specify your C6711 when you define the object properties `boardnum` and `procnum`.

Before continuing with this tutorial, you must load a valid GEL file to configure the EMIF registers of your processor and perform any required processor initialization steps. Default GEL files provided by CCS are stored in `..\cc\gel` in the folder where you installed CCS software. Select **File** > **Load GEL** in CCS IDE to load the default GEL file that matches your processor family, such as `init6x0x.gel` for the C6x0x processor family, and your configuration.

Begin the process of getting your model onto the processor by creating an object that refers to CCS IDE. Start by clearing all existing handles and setting echo on so you see functions in the M-file execute as the program runs:

```
1 clear all; echo on;
```

`clear all` has the side effect of removing debugging breakpoints and resetting persistent variables because function breakpoints and persistent variables are cleared whenever the M-file changes or is cleared. Breakpoints within your executable remain after `clear`. Clearing the MATLAB workspace does not affect your executable.

2 Now construct the link to your board and processor by entering

```
cc=ticcs('boardnum',0);
```

boardnum defines which board the new link accesses. In this example, boardnum is 0. Embedded IDE Link connects the link to the first, and in this case only, processor on the board. To find the boardnum and procnum values for the boards and simulators on your system, use ccsboardinfo. When you enter the following command at the prompt

```
ccsboardinfo
```

Embedded IDE Link returns a list like the following one that identifies the boards and processors in your computer.

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Inst...	0	CPU	TMS320C6211
0	C6701 EVM (Texas Instruments)	0	CPU_1	TMS320C6701

- 3 To open and load the processor file, change the path for MATLAB software to be able to find the file.

```
projname =
C:\Temp\LinkForCCSDemos_3.2\rtdxtutorial\c6x\c64xp\rtdxtut_sim.pjt

outFile =
C:\Temp\LinkForCCSDemos_v3.2\rtdxtutorial\c6x\c64xp\rtdxtut_sim.out

processor_dir = demoPjt.DemoDir

processor_dir =
C:\Temp\LinkForCCSDemos_v3.2\rtdxtutorial\c6x\c64xp
```

```
% Go to processor directory
cd(cc,processor_dir);cd(cc,tgt_dir); % Or cc.cd(tgt_dir)
dir(cc); % Or cc.dir
```

To load the appropriate project file to your processor, enter the following commands at the MATLAB software prompt. `getDemoProject` is a specialized function for loading Embedded IDE Link demo files. It is not supported as a standard Embedded IDE Link function.

```
demoPjt = getDemoProject(cc,'ccstutorial');

demoPjt.ProjectFile

ans =

C:\Temp\LinkForCCSDemos_v3.2\ccstutorial\c6x\c64xp\ccstut.pjt

demoPjt.DemoDir

ans =

C:\Temp\LinkForCCSDemos_v3.2\ccstutorial\c6x\c64xp
```

Notice where the demo files are stored on your machine. In general, Embedded IDE Link software stores the demo project files in

```
LinkforCCS_vproduct_version
```

For example, if you are using version 3.2 of Embedded IDE Link software, the project demos are stored in `LinkforCCS_v3.2\`. Embedded IDE Link software creates this folder in a location on your machine where you have write permission. Usually, there are two locations where Embedded IDE Link software tries to create the demo folder, in the order shown.

- a** In a temporary folder on the C drive, such as `C:\temp\`.
- b** If Embedded IDE Link software cannot use the `temp` folder, you see a dialog box that asks you to select a location to store the demos.

- 4 You have reset the folder path to find the tutorial file. Now open the .out file that matches your processor type, such as `rtdxtutorial_c67x.out` or `rtdxtutorial_c64x.out`.

```
cc.open('rtdxtutorial_67x.out')
```

Because `open` is overloaded for the CCS IDE and RTDX links, this may seem a bit strange. In this syntax, `open` loads your executable file onto the processor identified by `cc`. Later in this tutorial, you use `open` with a different syntax to open channels in RTDX.

In the next section, you use the new link to open and enable communications between MATLAB software and your processor.

Configuring Communications Channels

Communications channels to the processor do not exist until you open and enable them through Embedded IDE Link and CCS IDE. Opening channels consists of opening and configuring each channel for reading or writing, and enabling the channels.

In the `open` function, you provide the channel names as strings for the channel name property. The channel name you use is not random. The channel name string must match a channel defined in the executable file. If you specify a string that does not identify an existing channel in the executable, the `open` operation fails.

In this tutorial, two channels exist on the processor — `ichan` and `ochan`. Although the channels are named `ichan` for input channel and `ochan` for output channel, neither channel is configured for input or output until you configure them from MATLAB software or CCS IDE. You could configure `ichan` as the output channel and `ochan` as the input channel. The links would work just the same. For simplicity, the tutorial configures `ichan` for input and `ochan` for output. One more note—reading and writing are defined as seen by the processor. When you write data from MATLAB software, you write to the channel that the processor reads, `ichan` in this case. Conversely, when you read from the processor, you read from `ochan`, the channel that the processor writes to:

- 1** Configure buffers in RTDX to store the data until MATLAB software can read it into your workspace. Often, MATLAB software cannot read data as quickly as the processor can write it to the channel.

```
cc.rtdx.configure(1024,4); % define 4 channels of 1024 bytes each
```

Channel buffers are optional. Adding them provides a measure of insurance that data gets from your processor to MATLAB software without getting lost.

- 2** Define one of the channels as a write channel. Use 'ichan' for the channel name and 'w' for the mode. Either 'w' or 'r' fits here, for write or read.

```
cc.rtdx.open('ichan','w');
```

- 3** Now enable the channel you opened.

```
cc.rtdx.enable('ichan');
```

- 4** Repeat steps 2 and 3 to prepare a read channel.

```
cc.rtdx.open('ochan','r');  
cc.rtdx.enable('ochan');
```

- 5** To use the new channels, enable RTDX by entering

```
cc.rtdx.enable;
```

You could do this step before you configure the channels — the order does not matter.

- 6** Reset the global time-out to 20s to provide a little room for error. `ticcs` applies a default timeout value of 10s. In some cases this may not be enough.

```
cc.rtdx.get('timeout')  
ans =  
    10  
cc.rtdx.set('timeout', 20); % Reset timeout = 20 seconds
```

- 7 Check that the `timeout` property value is now 20s and that your object has the correct configuration for the rest of the tutorial.

```
cc.rtdx
```

```
RTDX Object:
```

```
API version:      1.0
Default timeout:  20.00 secs
Open channels:    2
```

Running the Application

To this point you have been doing housekeeping functions that are common to any application you run on the processor. You load the processor, configure the communications, and set up other properties you need.

In this tutorial task, you use a specific application to demonstrate a few of the functions available in Embedded IDE Link that let you experiment with your application while you develop your prototype. To demonstrate the link for RTDX `readmat`, `readmsg`, and `writemsg` functions, you write data to your processor for processing, then read data from the processor after processing:

- 1 Restart the program you loaded on the processor. `restart` ensures the program counter (PC) is at the beginning of the executable code on the processor.

```
cc.restart
```

Restarting the processor does not start the program executing. You use `run` to start program execution.

- 2 Type `cc.run('run');`

Using `'run'` for the run mode tells the processor to continue to execute the loaded program continuously until it receives a halt directive. In this mode, control returns to MATLAB software so you can work in MATLAB software while the program runs. Other options for the mode are

- `'runtohalt'` — start to execute the program and wait to return control to MATLAB software until the process reaches a breakpoint or execution terminates.

- 'tohalt' — change the state of a running processor to 'runtohalt' and wait to return until the program halts. Use tohalt mode to stop the running processor cleanly.

- 3** Type the following functions to enable the write channel and verify that the enable takes effect.

```
cc.rtdx.enable('ichan');  
cc.rtdx.isenabled('ichan')
```

If MATLAB software responds `ans = 0` your channel is not enabled and you cannot proceed with the tutorial. Try to enable the channel again and verify the status.

- 4** Write some data to the processor. Check that you can write to the processor, then use `writemsg` to send the data. You do not need to enter the if-test code shown.

```
if cc.rtdx.iswritable('ichan'), % Used in a script application.  
    disp('writing to processor...') % Optional to display progress.  
    indata=1:10  
    cc.rtdx.writemsg('ichan', int16(indata))  
end % Used in scripts for channel testing.
```

The if statement simulates writing the data from within a MATLAB software script. The script uses `iswritable` to check that the input channel is functioning. If `iswritable` returns 0 the script would skip the write and exit the program, or respond in some way. When you are writing or reading data to your processor in a script or M-file, checking the status of the channels can help you avoid errors during execution.

As your application runs you may find it helpful to display progress messages. In this case, the program directed MATLAB software to print a message as it reads the data from the processor by adding the function

```
disp('writing to processor...')
```

Function `cc.rtdx.writemsg('ichan', int16(indata))` results in 20 messages stored on the processor. Here's how.

When you write `indata` to the processor, the following code running on the processor takes your input data from `ichan`, adds one to the values and copies the data to memory:

```
while ( !RTDX_isInputEnabled(&ichan) )

/* wait for channel enable from MATLAB */
RTDX_read( &ichan, recvd, sizeof(recvd) );
puts("\n\n Read Completed ");

for (j=1; j<=20; j++) {
    for (i=0; i<MAX; i++) {
        recvd[i] +=1;
    }
    while ( !RTDX_isOutputEnabled(&ochan) )
        { /* wait for channel enable from MATLAB */ }
    RTDX_write( &ochan, recvd, sizeof(recvd) );
    while ( RTDX_writing != NULL )
        { /* wait for data xfer INTERRUPT DRIVEN for C6000 */ }
}
```

Program `int16_rtdx.c` contains this source code. You can find the file in a folder in the `..\tidemos\rtdxtutorial` folder.

- 5** Type the following to check the number of available messages to read from the processor.

```
num_of_msgs = cc.rtdx.msgcount('ochan');
```

`num_of_msgs` should be zero. Using this process to check the amount of data can make your reads more reliable by letting you or your program know how much data to expect.

- 6** Type the following to verify that your read channel `ochan` is enabled for communications.

```
cc.rtdx.isenabled('ochan')
```

You should get back `ans = 0` — you have not enabled the channel yet.

- 7** Now enable and verify `'ochan'`.

```
cc.rtdx.enable('ochan');  
cc.rtdx.isenabled('ochan')
```

To show that ochan is ready, MATLAB software responds `ans = 1`. If not, try enabling ochan again.

8 Type

```
pause(5);
```

The `pause` function gives the processor extra time to process the data in `indata` and transfer the data to the buffer you configured for ochan.

9 Repeat the check for the number of messages in the queue. There should be 20 messages available in the buffer.

```
num_of_msgs = cc.rtdx.msgcount('ochan')
```

With `num_of_msgs = 20`, you could use a looping structure to read the messages from the queue in to MATLAB software. In the next few steps of this tutorial you read data from the ochan queue to different data formats within MATLAB software.

10 Read one message from the queue into variable `outdata`.

```
outdata = cc.rtdx.readmsg('ochan','int16')
```

```
outdata =  
     2     3     4     5     6     7     8     9    10    11
```

Notice the `'int16'` represent option. When you read data from your processor you need to tell MATLAB software the data type you are reading. You wrote the data in step 4 as 16-bit integers so you use the same data type here.

While performing reads and writes, your process continues to run. You did not need to stop the processor to get the data or send the data, unlike using most debuggers and breakpoints in your code. You placed your data in memory across an RTDX channel, the processor used the data, and you read the data from memory across an RTDX channel, without stopping the processor.

- 11** You can read data into cell arrays, rather than into simple double-precision variables. Use the following function to read three messages to cell array `outdata`, an array of three, 1-by-10 vectors. Each message is a 1-by-10 vector stored on the processor.

```
outdata = cc.rtdx.readmsg('ochan','int16',3)

outdata =
 [1x10 int16] [1x10 int16] [1x10 int16]
```

- 12** Cell array `outdata` contains three messages. Look at the second message, or matrix, in `outdata` by using dereferencing with the array.

```
outdata{1,2}

outdata =
     4     5     6     7     8     9    10    11    12    13
```

- 13** Read two messages from the processor into two 2-by-5 matrices in your MATLAB workspace.

```
outdata = cc.rtdx.readmsg('ochan','int16',[2 5],2)

outdata =
 [2x5 int16] [2x5 int16]
```

To specify the number of messages to read and the data format in your workspace, you used the `siz` and `nummsgs` options set to `[2 5]` and `2`.

- 14** You can look at both matrices in `outdata` by dereferencing the cell array again.

```
outdata{1,:}

ans =
     6     8    10    12    14
     7     9    11    13    15
ans =
     7     9    11    13    15
     8    10    12    14    16
```

- 15** For a change, read a message from the queue into a column vector.

```
outdata = cc.rtdx.readmsg('ochan','int16',[10 1])
```

```
outdata =  
      8  
      9  
     10  
     11  
     12  
     13  
     14  
     15  
     16  
     17
```

- 16** Embedded IDE Link provides a function for reading messages into matrices—`readmat`. Use `readmat` to read a message into a 5-by-2 matrix in MATLAB software.

```
outdata = readmat(cc.rtdx,'ochan','int16',[5 2])
```

```
outdata =  
      9  14  
     10  15  
     11  16  
     12  17  
     13  18
```

Because a 5-by-2 matrix requires ten elements, MATLAB software reads one message into `outdata` to fill the matrix.

- 17** To check your progress, see how many messages remain in the queue. You have read eight messages from the queue so 12 should remain.

```
num_of_msgs = cc.rtdx.msgcount('ochan')
```

```
num_of_msgs =  
      12
```


- 18** To demonstrate the connection between messages and a matrix in MATLAB software, read data from 'ochan' to fill a 4-by-5 matrix in your workspace.

```
outdata = cc.rtdx.readmat('ochan','int16',[4 5])
```

```
outdata =  
    10    14    18    13    17  
    11    15    19    14    18  
    12    16    11    15    19  
    13    17    12    16    20
```

Filling the matrix required two messages worth of data.

- 19** To verify that the last step used two messages, recheck the message count. You should find 10 messages waiting in the queue.

```
num_of_msgs = cc.rtdx.msgcount('ochan')
```

- 20** Continuing with matrix reads, fill a 10-by-5 matrix (50 matrix elements or five messages).

```
outdata = cc.rtdx.readmat('ochan','int16',[10 5])
```

```
outdata =  
    12    13    14    15    16  
    13    14    15    16    17  
    14    15    16    17    18  
    15    16    14    18    19  
    16    17    18    19    20  
    17    18    19    20    21  
    18    19    20    21    22  
    19    20    21    22    23  
    20    21    22    23    24  
    21    22    23    24    25
```

- 21** Recheck the number of messages in the queue to see that five remain.
- 22** flush lets you remove messages from the queue without reading them. Data in the message you remove is lost. Use flush to remove the next message in the read queue. Then check the waiting message count.

```
cc.rtdx.flush('ochan',1)
num_of_msgs = cc.rtdx.msgcount('ochan')

num_of_msgs =

4
```

- 23** Empty the remaining messages from the queue and verify that the queue is empty.

```
cc.rtdx.flush('ochan','all')
```

With the `all` option, `flush` discards all messages in the `ochan` queue.

Closing the Connections and Channels or Cleaning Up

One of the most important programmatic processes you should do in every RTDX session is to clean up at the end. Cleaning up includes stopping your processor, disabling the RTDX channels you enabled, disabling RTDX and closing your open channels. Performing this series of tasks ensures that future processes avoid trouble caused by unexpected interactions with remaining handles, channels, and links from earlier development work.

Best practices suggest that you include the following tasks (or an appropriate subset that meets your development needs) in your development scripts and programs.

We use several functions in this section; each has a purpose — the operational details in the following list explain how and why we use each one. They are

- **close** — close the specified RTDX channel. To use the channel again, you must open and enable the channel. Compare **close** to **disable**. `close('rtdx')` closes the communications provided by RTDX. After you close RTDX, you cannot communicate with your processor.
- **disable** — remove RTDX communications from the specified channel, but does not remove the channel, or link. Disabling channels may be useful when you do not want to see the data that is being fed to the channel, but you may want to read the channel later. By enabling the channel later, you have access to the data entering the channel buffer. Note that data that entered the channel while it was disabled is lost.

- `halt` — stop a running processor. You may still have one or more messages in the host buffer.

Use the following procedure to shut down communications between MATLAB software and the processor, and end your session:

- 1 Begin the process of shutting down the processor and RTDX by stopping the processor. Type the following functions at the prompt.

```
if (isrunning(cc)) % Use this test in scripts.
    cc.halt;        % Halt the processor.
end                % Done.
```

Your processor may already be stopped at this point. In a script, you might put the function in an `if`-statement as we have done here. Consider this test to be a safety check. No harm comes to the processor if it is already stopped when you tell it to stop. When you direct a stopped processor to `halt`, the function returns immediately.

- 2 You have stopped the processor. Now disable the RTDX channels you opened to communicate with the processor.

```
cc.rtdx.disable('all');
```

If necessary, using `disable` with channel name and processor identifier input arguments lets you disable only the channel you choose. When you have more than one board or processor, you may find disabling selected channels meets your needs.

When you finish your RTDX communications session, disable RTDX to ensure that Embedded IDE Link releases your open channels before you close them.

```
cc.rtdx.disable;
```

- 3 Use one or all of the following function syntaxes to close your open channels. Either close selected channels by using the channel name in the function, or use the `all` option to close all open channels.
 - `cc.rtdx.close('ichan')` to close your input channel in this tutorial.
 - `cc.rtdx.close('ochan')` to close your output channel in the tutorial.

- `cc.rtdx.close('all')` to close all of your open RTDX channels, regardless of whether they are part of this tutorial.

Consider using the `all` option with the `close` function when you finish your RTDX work. Closing channels reduces unforeseen problems caused by channel objects that exist but do not get closed correctly when you end your session.

- 4 When you created your RTDX object (`cc = ticcs('boardnum',1)`) at the beginning of this tutorial, the `ticcs` function opened CCS IDE and set the visibility to 0. To avoid problems that occur when you close the interface to RTDX with CCS visibility set to 0, make CCS IDE visible on your desktop. The following `if` statement checks the CCS IDE visibility and changes it if needed.

```
if cc.isvisible,  
    cc.visible(1);  
end
```

Visibility can cause problems. When CCS IDE is running invisibly on your desktop, do not use `clear all` to remove your links for CCS IDE and RTDX. Without a `ticcs` object that references the CCS IDE you cannot access CCS IDE to change the visibility setting, or close the application. To close CCS IDE when you do not have an existing object, either create a new object to access the CCS IDE, or use Microsoft Windows Task Manager to end the process `cc_app.exe`, or close the MATLAB software.

- 5 You have finished the work in this tutorial. Enter the following commands to close your remaining references to CCS IDE and release the associated resources.

```
clear ('all'); % Calls the link destructors to remove all links.  
echo off
```

`clear all` without the parentheses removes all variables from your MATLAB workspace.

You have completed the tutorial using RTDX. During the tutorial you

- 1 Opened connections to CCS IDE and RTDX and used those connections to load an executable program to your processor.

- 2** Configured a pair of channels so you could transfer data to and from your processor.
- 3** Ran the executable on the processor, sending data to the processor for processing and retrieving the results.
- 4** Stopped the executing program and closed the links to CCS IDE and RTDX.

This tutorial provides a working process for using Embedded IDE Link and your signal processing programs to develop programs for a range of Texas Instruments processors. While the processor may change, the essentials of the process remain the same.

Listing Functions

To review a complete list of functions and methods that operate with `ticcs` objects, either CCS IDE or RTDX, enter either of the following commands at the prompt.

```
help ticcs
help rtdx
```

If you already have a `ticcs` object `cc`, you can use dot notation to return the methods for CCS IDE or RTDX by entering one of the following commands at the prompt:

- `cc.methods`
- `cc.rtdx.methods`

In either instance MATLAB software returns a list of the available functions for the specified link type, including both public and private functions. For example, to see the functions (methods) for links to CCS IDE, enter:

```
help ticcs
```

Constructing ticcs Objects

When you create a connection to CCS IDE using the `ticcs` command, you are creating a “`ticcs` object for accessing the CCS IDE and RTDX interface”. The `ticcs` object implementation relies on MATLAB software object-oriented programming capabilities.

The discussions in this section apply to the `ticcs` objects in Embedded IDE Link.

Like other MATLAB software structures, objects in Embedded IDE Link have predefined fields called object properties.

You specify object property values by one of the following methods:

- Setting the property values when you create the `ticcs` object
- Creating an object with default property values, and changing some or all of these property values later

For examples of setting `ticcs` object properties, refer to `ticcs`.

Example – Constructor for ticcs Objects

The easiest way to create an object is to use the function `ticcs` to create an object with the default properties. Create an object named `cc` to refer to CCS IDE by entering

```
cc = ticcs
```

MATLAB software responds with a list of the properties of the object `cc` you created along with the associated default property values.

```
ticcs object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs
```

```
RTDX channels      : 0
```

Inspecting the output reveals two objects listed—a CCS IDE object and an RTDX object. CCS IDE and RTDX objects cannot be created separately. By design they maintain a member class relationship; the RTDX object is a class, a member of the CCS object class. In this example, `cc` is an instance of the class CCS. If you enter

```
rx = cc.rtdx
```

`rx` is a handle to the RTDX portion of the CCS object. As an alias, `rx` replaces `cc.rtdx` in functions such as `readmat` or `writemsg` that use the RTDX communications features of the CCS link. Typing `rx` at the command line now produces

```
rx
```

```
RTDX channels      : 0
```

The object properties are described in Chapter 6, “Function Reference”, and in more detail in ticcs Object Properties. These properties are set to default values when you construct objects.

tics Properties and Property Values

Objects in Embedded IDE Link software have properties associated with them. Each property is assigned a value. You can set the values of most properties, either when you create the link or by changing the property value later. However, some properties have read-only values. And a few property values, such as the board number and the processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

For more information about using objects and properties, refer to “Using Objects” in *MATLAB Programming Fundamentals*.

Overloaded Functions for ticcs Objects

Several functions in this Embedded IDE Link have the same name as functions in other MathWorks toolboxes or in MATLAB software. These behave similarly to their original counterparts, but you apply these functions directly to an object. This concept of having functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `set` command is overloaded for `ticcs` objects. After you specify your link by assigning values to its properties, you can apply the functions in this toolbox (such as `readmat` for using RTDX to read an array of data from the processor) directly to the variable name you assign to your object, without specifying your object parameters again.

For a complete list of the functions that act on `ticcs` objects, refer to the tables of functions in the function reference pages.

ticcs Object Properties

In this section...
“Quick Reference to ticcs Object Properties” on page 2-46
“Details About ticcs Object Properties” on page 2-48

Embedded IDE Link provides an interface to your processor hardware so you can communicate with processors for which you are developing systems and algorithms. Each ticcs object comprises two objects—a CCS IDE object and an RTDX interface object. The objects are not separable; the RTDX object is a subclass of the CCS IDE object. Each of the objects has multiple properties. To configure the interface objects for CCS IDE and RTDX, you set parameters that define details such as the desired board, the processor, the timeout period applied for communications operations, and a number of other values. Because Embedded IDE Link uses objects to create the interface, the parameters you set are called properties and you treat them as properties when you set them, retrieve them, or modify them.

This section details the properties for the ticcs objects for CCS IDE and RTDX. First the section provides tables of the properties, for quick reference. Following the tables, the section offers in-depth descriptions of each property, its name and use, and whether you can set and get the property value associated with the property. Descriptions include a few examples of the property in use.

MATLAB software users may find much of this handling of objects familiar. Objects in Embedded IDE Link, behave like objects in MATLAB software and the other object-oriented toolboxes. For C++ programmers, discussion of object-oriented programming is likely to be a review.

Quick Reference to ticcs Object Properties

The following table lists the properties for the ticcs objects in Embedded IDE Link. The second column tells you which object the property belongs to. Knowing which property belongs to each object in a ticcs object tells you how to access the property.

Property Name	Applies to Which Connection?	User Settable?	Description
apiversion	CCS IDE	No	Reports the version number of your CCS API.
boardnum	CCS IDE	Yes/initially	Specifies the index number of a board that CCS IDE recognizes.
ccsappexe	CCS IDE	No	Specifies the path to the CCS IDE executable. Read-only property.
numchannels	RTDX	No	Contains the number of open RTDX channels for a specific link.
page	CCS IDE	Yes/default	Stores the default memory page for reads and writes.
procnum	CCS IDE	Yes/at start only	Stores the number CCS Setup Utility assigns to the processor.
rtdx	RTDX	No	Specifies RTDX in a syntax.
rtdxchannel	RTDX	No	A string. Identifies the RTDX channel for a link.
timeout	CCS IDE	Yes/default	Contains the global timeout setting for the link.
version	RTDX	No	Reports the version of your RTDX software.

Some properties are read only — you cannot set the property value. Other properties you can change at all times. If the entry in the User Settable column is “Yes/initially”, you can set the property value only when you create the link. Thereafter it is read only.

Details About `ticc` Object Properties

To use the links for CCS IDE and RTDX interface you set values for:

- `boardnum` — specify the board with which the link communicates.
- `procnum` — specify the processor on the board. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — specify the global timeout value. (Optional. Default is 10s.)

Details of the properties associated with connections to CCS IDE and RTDX interface appear in the following sections, listed in alphabetical order by property name.

Many of these properties are object linking and embedding (OLE) handles. The MATLAB software COM server creates the handles when you create objects for CCS IDE and RTDX. You can manipulate the OLE handles using `get`, `set`, and `invoke` to work directly with the COM interface with which the handles interact.

apiversion

Property `apiversion` contains a string that reports the version of the application program interface (API) for CCS IDE that you are using when you create a link. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `apiversion` property value for a link. This example shows the `apiversion` value for link `cc`.

```
display(cc)
```

```
TICCS Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0
```

Note that the API version is not the same as the CCS IDE version.

boardnum

Property `boardnum` identifies the board referenced by a link for CCS IDE. When you create a link, you use `boardnum` to specify the board you are using. To get the value for `boardnum`, use `ccsboardinfo` or the CCS Setup utility from Texas Instruments software. The CCS Setup utility assigns the number for each board installed on your system.

ccsappexe

Property `ccsappexe` contains the path to the CCS IDE executable file `cc_app.exe`. When you use `ticcs` to create a link, MATLAB software determines the path to the CCS IDE executable and stores the path in this property. This is a read-only property. You cannot set it.

numchannels

Property `numchannels` reports the number of open RTDX communications channels for an RTDX link. Each time you open a channel for a link, `numchannels` increments by one. For new links `numchannels` is zero until you open a channel for the link.

To see the value for `numchannels` create a link to CCS IDE. Then open a channel to RTDX. Use `display` to see the RTDX link properties.

```
cc=ticcs
```

```
TICCS Object:
```

```
API version      : 1.0
Processor type   : C67
Processor name   : CPU
Running?        : No
Board number     : 0
Processor number : 0
Default timeout  : 10.00 secs

RTDX channels    : 0
```

```
rx=cc.rtdx

    RTDX channels      : 0

open(rx,'ichan','r','ochan','w');

get(cc.rtdx)

ans =

    numChannels: 2
           Rtdx: [1x1 COM ]
    RtdxChannel: {' ' ' '}
           procType: 103
           timeout: 10
```

page

Property `page` contains the default value CCS IDE uses when the user does not specify the `page` input argument in the syntax for a function that access memory.

procnum

Property `procnum` identifies the processor referenced by a link for CCS IDE. When you create an object, you use `procnum` to specify the processor you are using. The CCS Setup Utility assigns a number to each processor installed on each board. To determine the value of `procnum` for a processor, use `ccsboardinfo` or the CCS Setup utility from Texas Instruments software.

To identify a processor, you need both the `boardnum` and `procnum` values. For boards with one processor, `procnum` equals zero. CCS IDE numbers the processors on multiprocessor boards sequentially from 0 to the number of processors. For example, on a board with four processors, the processors are numbered 0, 1, 2, and 3.

rtdx

Property `rtdx` is a subclass of the `ticcs` link and represents the RTDX portion of a link for CCS IDE. As shown in the example, `rtdx` has properties of its own that you can set, such as `timeout`, and that report various states of the link.

```
get(cc.rtdx)

ans =

    version: 1
  numChannels: 0
      Rtdx: [1x1 COM ]
RtdxChannel: {'' [] ''}
    procType: 103
      timeout: 10
```

In addition, you can create an alias to the `rtdx` portion of a link, as shown in this code example.

```
rx=cc.rtdx

RTDX channels : 0
```

Now you can use `rx` with the functions in Embedded IDE Link, such as `get` or `set`. If you have two open channels, the display looks like the following

```
get(rx)

ans =

    numChannels: 2
      Rtdx: [1x1 COM ]
RtdxChannel: {2x3 cell}
    procType: 98
      timeout: 10
```

when the processor is from the C62 family.

rtdxchannel

Property `rtdxchannel`, along with `numchannels` and `proctype`, is a read-only property for the RTDX portion of a link for CCS IDE. To see the value of this property, use `get` with the link. Neither `set` nor `invoke` work with `rtdxchannel`.

`rtdxchannel` is a cell array that contains the channel name, handle, and mode for each open channel for the link. For each open channel, `rtdxchannel` contains three fields, as follows:

<code>.rtdxchannel{i,1}</code>	Channel name of the <i>i</i> th-channel, <i>i</i> from 1 to the number of open channels
<code>.rtdxchannel{i,2}</code>	Handle for the <i>i</i> th-channel
<code>.rtdxchannel{i,3}</code>	Mode of the <i>i</i> th-channel, either 'r' for read or 'w' for write

With four open channels, `rtdxchannel` contains four channel elements and three fields for each channel element.

timeout

Property `timeout` specifies how long CCS IDE waits for any process to finish. Two `timeout` periods can exist — one global, one local. You set the global `timeout` when you create a link for CCS IDE. The default global `timeout` value 10 s. However, when you use functions to read or write data to CCS IDE or your processor, you can set a local `timeout` that overrides the global value. If you do not set a specific `timeout` value in a read or write process syntax, the global `timeout` value applies to the operation. Refer to the help for the read and write functions for the syntax to set the local `timeout` value for an operation.

version

Property `version` reports the version number of your RTDX software. When you create a `ticcs` object, `version` contains a string that reports the version of the RTDX application that you are using. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `version` property value for a link. This example shows the `apiversion` value for object `rx`.


```
get(rx) % rx is an alias for cc.rtdx.
```

```
ans =
```

```
    version: 1  
 numChannels: 0  
      Rtdx: [1x1 COM ]  
RtdxChannel: {'' [] ''}  
   procType: 103  
   timeout: 10
```

Managing Custom Data Types with the Data Type Manager

Using custom data types, called typedefs (using the C keyword `typedef`), is one of the complications you encounter when you use hardware-in-the-loop (HIL) to run a function in your project from MATLAB. Because MATLAB does not recognize custom type definitions you use in your projects, it cannot interpret data that you define in your project code with the `typedef` keyword, or use as arguments in your function prototype (declaration).

To allow you to use functions that include custom type definitions in function calls, Embedded IDE Link offers the Data Type Manager (DTM), a tool for defining custom type definitions to MATLAB. Using options in the DTM, you define one or more custom data types for a project and use them in the project. Or you define your custom data types and save them to use in many projects. This second feature is particularly useful when you use the same custom data types in many projects. Rather than redefining your custom types for each new project or function, you reload the types from an earlier project to use them again.

As programmers, usually you use typedefs for one or more of a few reasons:

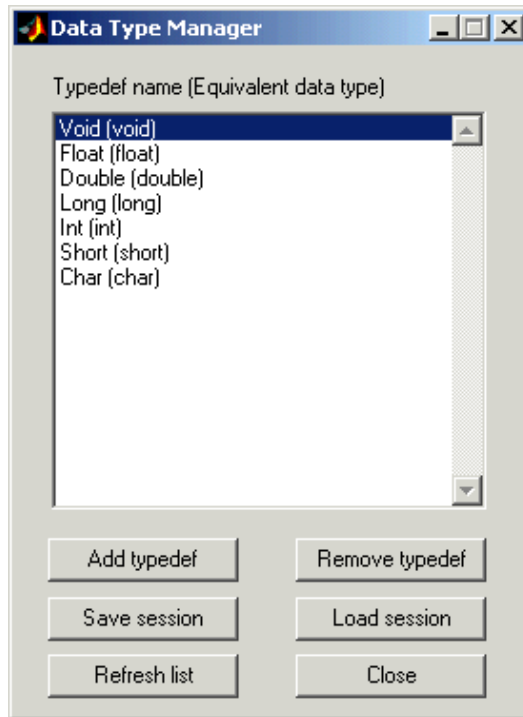
- Make your code more accessible by providing more information about the variable(s)
- Create a Boolean data type that C does not provide
- Define structures in your programs
- Define nonstandard data types

The DTM lets you define all of these things in the MATLAB context so your C function that uses typedefs works with your MATLAB command line functions. For reference information about the DTM, go to [datatypemanager](#).

Entering

```
datatypemanager(cc)
```

at the MATLAB command line opens the DTM, with the Data Type Manager dialog box shown here:



When the DTM opens, a variety of information and options displays in the Data Type Manager dialog box:

- **Typedef name (Equivalent data type)** — provides a list of default data types. When you create a typedef, you see it added to this list.

The lowercase versions of the data types appear because MATLAB does not recognize the initial capital versions automatically. In the data type list the project data type with the initial capital letter is mapped to the lowercase MATLAB data type.

- **Add typedef** — opens the **Add Typedef** dialog box so you can add one or more typedefs to your object. Your added typedef appears on the **Typedef name (Equivalent data type)** list and is added to your ticcs object. Also,

when you pass the `cc` object to the DTM, and then add a `typedef`, the command

```
cc.type
```

returns the list of data types in the `type` property of your `cc` object, including the `typedefs` you added.

- **Remove typedef** — removes a selected `typedef` from the **Typedef name (Equivalent data type)** list.
- **Load session** — loads a previously saved session so you can use the `typedefs` you defined earlier without reentering them.
- **Refresh list** — updates the list in **Typedefs name (Equivalent data type)**. Refreshing the list ensures the contents are current. If you changed your project data type content or loaded a new project, this updates the type definitions in the DTM.
- **Close** — closes the DTM and prompts you to save the session information. This is the only way to save your work in this dialog box. Saving the session creates an M-file you can reload into the DTM later.

Adding Custom Type Definitions to MATLAB

Every custom type definition in your project must appear on the **Typedef name (Equivalent data type)** list for MATLAB to understand the data types involved. To add entries the list, use the **Add typedef** option to identify your type definition with a data type that MATLAB recognizes. When you click **Add typedef**, the **List of Known Data Types** dialog box opens, displaying the data types currently recognized by MATLAB. To make finding a specific type easier, the known data types are grouped into categories:

- MATLAB types
- TI C types
- TI fixed point types
- Struct, union, enum types
- Other (e.g. pointers, `typedefs`)

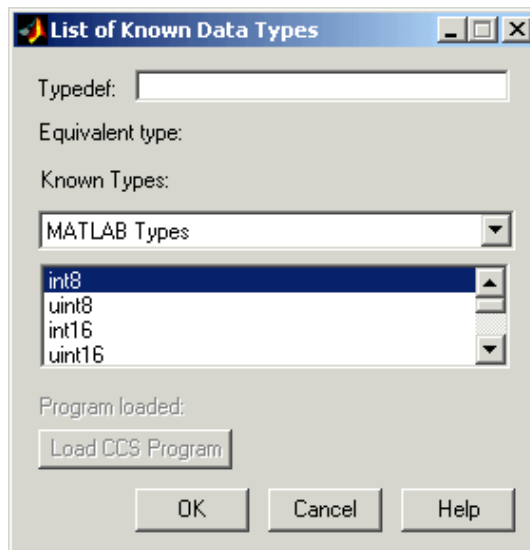
Each custom type definition added in the DTM becomes part of the `ticcs` object passed to the DTM in `datatypemanager(objectname)`. The list of data types in the object, both default and custom, is available by entering

```
objectname.type
```

at the command prompt.

The same list appears in the DTM on the **Typedef name (Equivalent data type)**

MATLAB uses the type definitions when you run a function residing on your processor from MATLAB.



To Add a Typedef to MATLAB

You use the DTM to add typedefs for MATLAB to recognize, such as:

- Typedefs that use a MATLAB data type in the type definition
- Typedefs that use an enumerated or union data type in the type definition
- Typedefs that use a structure in the type definition

- Typedefs that use pointers or typedefs in the type definition

To define custom data types that use structs, enums, or unions from a project, the project must be loaded on the processor before you add the custom type definitions. Either load the project and .out file before you start the DTM, or use the **Load Program** option in the DTM to load the .out file.

Note After a successful load process, you see the name of the file you loaded in **Loaded program**. Otherwise, you get an error message that the load failed.

Only programs that you load from this dialog box appear in **Program loaded**. Programs that are already loaded on your processor do not appear there because MATLAB cannot determine what program you have loaded.

You need to know the custom definitions you used so you can add them in the DTM. Use the options for **list** to verify whether you loaded a .out file on the processor.

Create an object and load a program.

- 1 Create a ticcs object.

```
cc=ticcs;
```

- 2 Load a program on your processor. For example, the MATLAB command

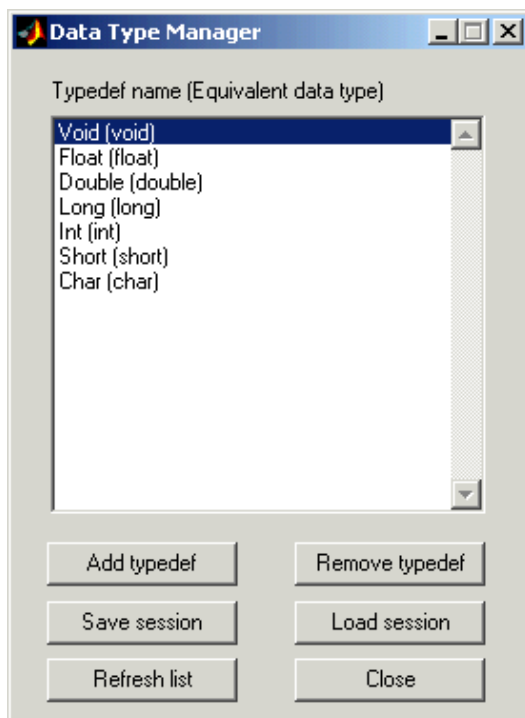
```
load(cc, 'c6711dskwdnoisf_c6000_rtwD\c6711dskwdnoisf.out');
```

loads the executable file from the model c6711dskwdnois.mdl on the processor.

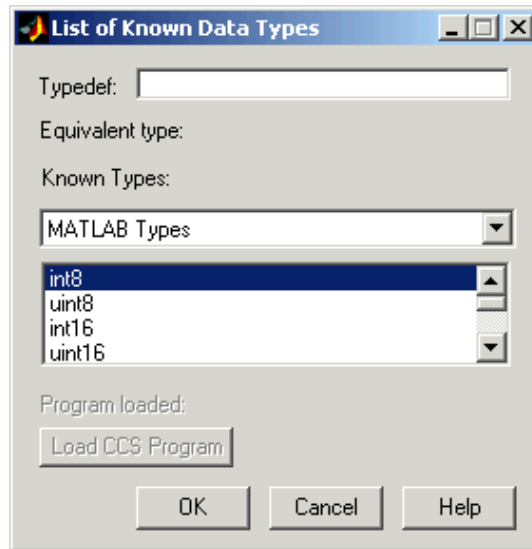
- 3 Start the DTM with the object you created.

```
datatypemanager(cc);
```

The DTM starts, showing the default data types.



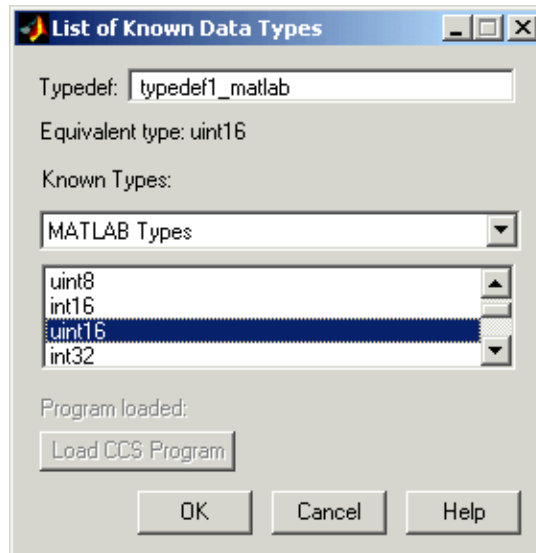
- 4** Click **Add typedef** to add your first custom data type. The List of Known Data Types dialog box appears as shown.



Add a MATLAB type definition.

- 5** In **Typedef**, enter the name of the typedef as you defined it in your code. For this example, use typedef1_matlab.

- 6 Select an appropriate MATLAB data type from the MATLAB Types in **Known Types**. `uint16` is the choice. Choose the data type that best represents the data type in your code.

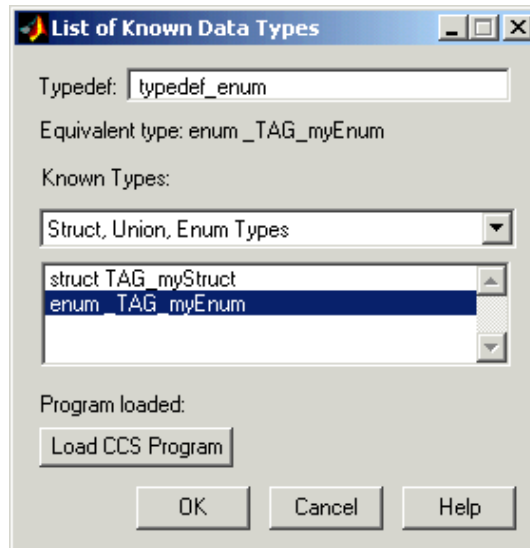


- 7 Click **OK** to close the dialog box and add the new type definition to the **Typedef name** list.

Add an enumerated type definition.

- 8 Click **Add Typedef**.
- 9 From the **Known Types** list, select Struct, Enum, Union Types.
- 10 To define your type definition, give it a name in **Typedef**, such as `typedef_enum`

- 11 From the Struct, Enum, Union Types list, select the appropriate enumerated data type to use with `typedef_enum`. The `enum_TAG_myEnum` choice fills the enumerated type chosen.

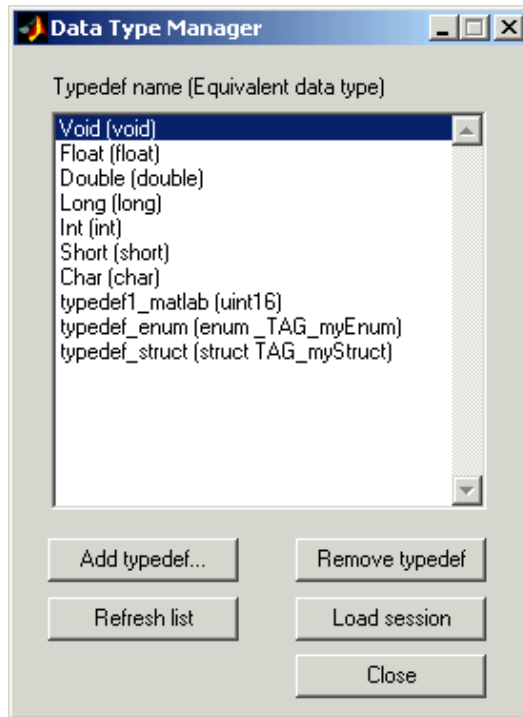


- 12 Click **OK** to close the dialog box and add `typedef_enum` to your defined types that MATLAB software recognizes.

Add a structure typedef.

- 13 Click **Add Typedef**.
- 14 From the **Known Types** list, select Struct, Enum, Union Types.
- 15 To define your type definition, give it a name in **Typedef**, such as `typedef_struct`.
- 16 From the Struct, Enum, Union Types list, select the appropriate enumerated data type to use with `typedef_struct`. This example uses `struct_TAG_myStruct`.
- 17 Click **OK** to close the dialog box and add the new data type to the list.

After you close the dialog box, the **Typedef name** list in the Data Type Manager looks like this.



To check the data types in the `cc` object, enter

```
cc.type
```

which returns

```
Defined types      : Void, Float, Double, Long, Int, Short, Char,  
typedef1_matlab, typedef_enum, typedef_struct
```

If your function declaration uses any of the types listed by `cc.type`, MATLAB software can interpret the data correctly. For example, MATLAB software interprets the `typedef1_matlab` data type as `uint16`.

Clicking **Close** in the DTM prompts you to save your session. Saving the session creates an M-file that contains operations that create your final list of data types, identical to the data types in the **Typedef name** list.

The first line of the M-file is a function definition, where the name of the function is the filename of the session you saved. In the stored M-file, you find a function that includes add and remove operations that replicate the add and remove `typedef` operations you used to create the list of known data types in the DTM. For each time you added a `typedef` in the DTM, the M-file contains an add command that adds the new type definition to the `type` property of the `cc` object. When you removed a data type, you created an equivalent `clear` command that removes the specified data type from the `type` property of the `cc` object.

All the operations you performed adding and removing data types in the DTM during the session are stored in the generated M-file that you save, including mistakes you made while creating or removing type definitions. When you load your saved session into the DTM, you see the same error messages you saw, during the session. Keep in mind that you have already corrected these errors.

Project Generator

- “Introducing Project Generator” on page 3-2
- “Project Generation and Board Selection” on page 3-3
- “Schedulers and Timing” on page 3-5
- “Project Generator Tutorial” on page 3-24
- “Setting Code Generation Parameters for TI Processors” on page 3-33
- “Setting Model Configuration Parameters” on page 3-36
- “Using Custom Source Files in Generated Projects” on page 3-48
- “Optimizing Embedded Code with Target Function Libraries” on page 3-52
- “Model Reference” on page 3-59

Introducing Project Generator

Project generator provides the following features for developing project and generating code:

- Support automated project building for Texas Instruments' Code Composer Studio software that lets you create projects from code generated by Real-Time Workshop and Real-Time Workshop Embedded Coder products. The project automatically populates CCS projects in the CCS development environment.
- Configure code generation using model configuration parameters and processor preferences block options
- Select from two system target files to generate code specific to your processor
- Configure project build process
- Automatically download and run your generated projects on your processor

Note You cannot generate code for C6000 processors in big-endian mode. Code generation supports only little-endian processor data byte order.

Project Generation and Board Selection

Project Generator uses `ticcs` objects to connect to the IDE. Each time you build a model to generate a project, the build process starts by issuing the `ticcs` method, as shown here:

```
cc=ticcs('boardnum',boardnum,'procnum',procnum)
```

The software attempts to connect to the board (`boardnum`) and processor (`procnum`) associated with the **Board name** and **Processor number** parameters in the Target Preferences block in the model.

The result of the `ticcs` method changes, depending on the boards you configured in CCS. The following table describes how the software selects the board to connect to in your board configuration.

CCS Board Configuration State	Response by Software
Code Composer Studio or Embedded IDE Link software not installed.	Returns an error message asking you to verify that you installed both Code Composer Studio and Embedded IDE Link properly.
Code Composer Studio software does not have any configured boards.	Returns an error message that the software could not find any boards in your configuration. Use Setup Code Composer Studio™ to configure at least one board.
Code Composer Studio software has one configured board.	Attaches to the board regardless of the name of the board supplied in the Target Preferences block. You see a warning message telling you which board the software selected.
Code Composer Studio software has one board configured that does not match the board name in the Target Preferences block. ^(*)	Returns a warning message that the software could not find the board specified in the block and connected to the board listed in the warning message. The software connects to the first board in your CCS configuration.

CCS Board Configuration State	Response by Software
Code Composer Studio has more than one board configured. The board name specified in the Target Preferences block is one of the configured boards.	Connects to the specified board.
Code Composer Studio has more than one board configured. The board name specified in the Target Preferences block is not one of the configured boards. ^(*)	<p>Returns a message asking you to select a board from the list of configured boards. You have two choices:</p> <ul style="list-style-type: none"> • Select a board to use for project generation, and click OK. Your selection does not change the board specified in the Target Preferences block. The software connects to the selected board. • Click Abort to stop the project build and code generation process. The software does not connect to the IDE or board.

^(*)You may encounter the situation where you do not have the correct board configured in CCS because of one of the following conditions:

- You changed your board configuration after you added the Target Preferences block to a model and saved the model. When you reopen the model, the board specified in **Board name** in the block is no longer in your configuration.
- You are working with a model from a source whose board configuration is not the same as yours. The model includes a Target Preferences block.

Use `ccsboardinfo` at the MATLAB prompt to verify or review your configured boards.

Schedulers and Timing

In this section...

- “Configuring Models for Asynchronous Scheduling” on page 3-5
- “Cases for Using Asynchronous Scheduling” on page 3-6
- “Comparing Synchronous and Asynchronous Interrupt Processing” on page 3-8
- “Using Synchronous Scheduling” on page 3-10
- “Using Asynchronous Scheduling” on page 3-10
- “Multitasking Scheduler Examples” on page 3-11

Configuring Models for Asynchronous Scheduling

Using the scheduling blocks, you can use an asynchronous (real-time) scheduler for your processor application. The asynchronous scheduler enables you to define interrupts and tasks to occur when you want by using blocks in the following block libraries:

- `idelinklib_common`

Note

- One way to view the block libraries is by entering the block library name at the MATLAB command line. For example: `>> idelinklib_common`
- You cannot build and run the models in following examples without additional blocks. They are for illustrative purposes only.

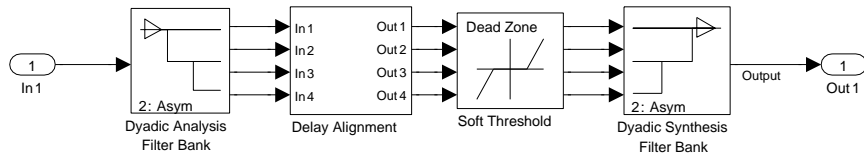
Also, you can schedule multiple tasks for asynchronous execution using the blocks.

The following figures show a model updated to use the asynchronous scheduler by converting the model to a function subsystem and then adding

a scheduling block (Hardware Interrupt) to drive the function subsystem in response to interrupts.

Before

The following model uses synchronous scheduling provided by the base rate in the model.

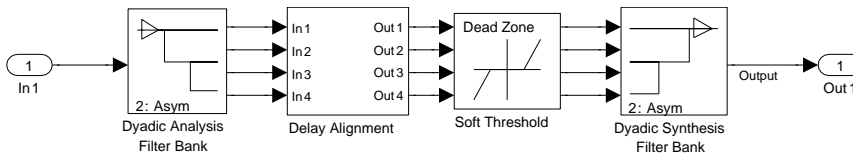


After

To convert to asynchronous operation, wrap the model in the previous figure in a function block and drive the input from a Hardware Interrupt block. The hardware interrupts that trigger the Hardware Interrupt block to activate an ISR now triggers the model inside the function block.

Algorithm Inside the Function Call Subsystem Block

Here's the model inside the function call subsystem in the previous figure. It is the same as the original model that used synchronous scheduling.

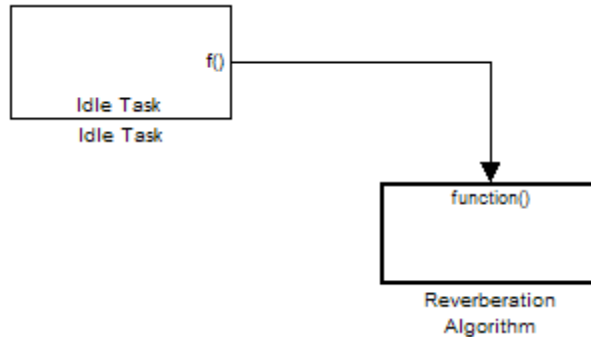


Cases for Using Asynchronous Scheduling

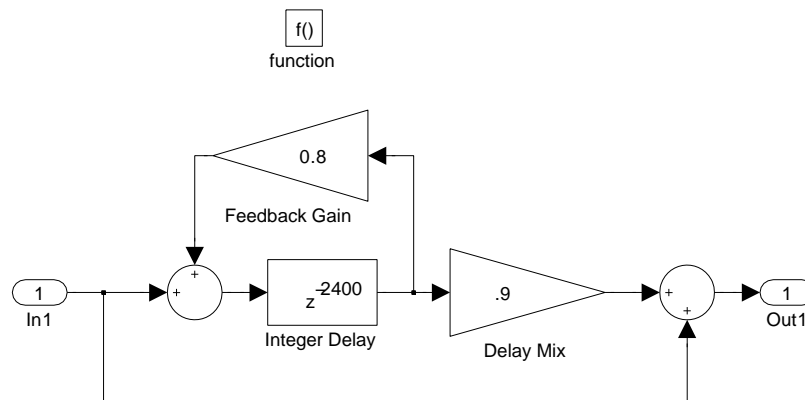
The following sections present common cases for using the scheduling blocks described in the previous sections.

Idle Task

The following model illustrates a case where the reverberation algorithm runs in the context of a background task in bare-board code generation mode.



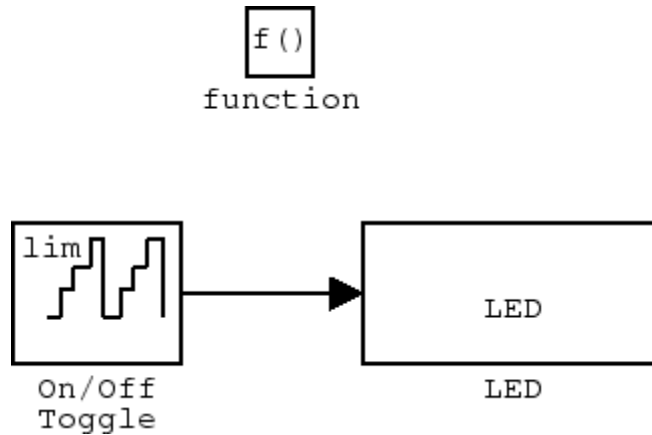
The function generated for this task normally runs in free-running mode—repetitively and indefinitely. Subsystem execution of the reverberation function is data driven via a background DMA interrupt-controlled ISR, shown in the following figure.



Hardware Interrupt Triggered Task

In the next figure, you see a case where a function (LED Control) runs in the context of a hardware interrupt triggered task.

In this model, the Hardware Interrupt block installs a task that runs when it detects an external interrupt. This task performs the specified function with an LED.

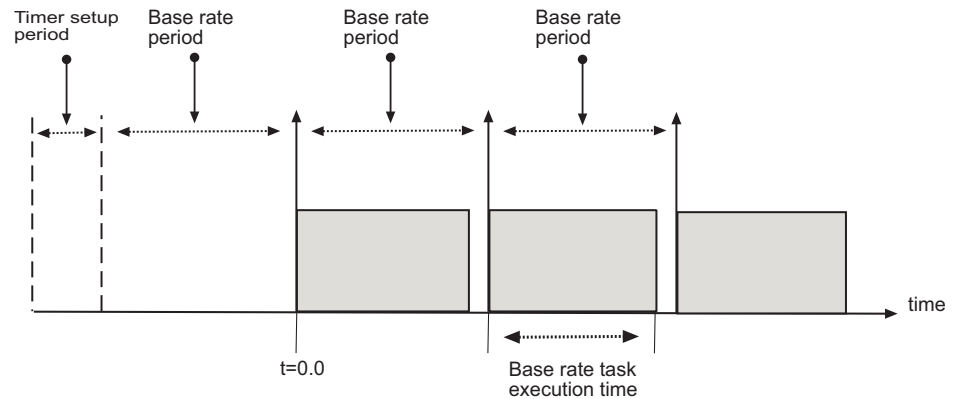


Comparing Synchronous and Asynchronous Interrupt Processing

Code generated for periodic tasks, both single- and multitasking, runs via a timer interrupt. A timer interrupt ensures that the generated code representing periodic-task model blocks runs at the specified period. The periodic interrupt clocks code execution at runtime. This periodic interrupt clock operates on a period equal to the base sample time of your model.

Note The execution of synchronous tasks in the model commences at the time of the first timer interrupt. Such interrupt occurs at the end of one full base rate period which follows timer setup. The time of the start of the execution corresponds to $t=0$.

The following figure shows the relationship between model startup and execution. Execution starts where your model executes the first interrupt, offset to the right of $t=0$ from the beginning of the time line. Before the first interrupt, the simulation goes through the timer set up period and one base rate period.



Timer-based scheduling does not provide enough flexibility for some systems. Systems for control and communications must respond to asynchronous events in real time. Such systems may need to handle a variety of hardware interrupts in an asynchronous, or *aperiodic*, fashion.

When you plan your project or algorithm, select your scheduling technique based on your application needs.

- If your application processes hardware interrupts asynchronously, add the appropriate asynchronous scheduling blocks from the library to your model:
 - A Hardware Interrupt block, to create an interrupt service routine to handle hardware interrupts on the selected processor
 - An Idle Task block, to create a task that runs as a separate thread
- Simulink sets the base rate priority to 40, the lowest priority.
- If your application does not service asynchronous interrupts, include only the algorithm and device driver blocks that specify the periodic sample times in the model.

Note Generating code from a model that does not service asynchronous interrupts automatically enables and manages a timer interrupt. The periodic timer interrupt clocks the entire model.

Using Synchronous Scheduling

Code that runs synchronously via a timer interrupt requires an interrupt service routine (ISR). Each model iteration runs after an ISR services a posted interrupt. The code generated for Embedded IDE Link uses a timer. To calculate the timer period, the software uses the following equation:

$$Timer_Period = \frac{(CPU_Clock_Rate) * (Base_Sample_Time)}{Low_Resolution_Clock_Divider} * Prescaler$$

The software configures the timer so that the base rate sample time for the coded process corresponds to the interrupt rate. Embedded IDE Link calculates and configures the timer period to ensure the desired sample rate.

Different processor families use the timer resource and interrupt number differently. Entries in the following table show the resources each family uses.

The minimum base rate sample time you can achieve depends on two factors—the algorithm complexity and the CPU clock speed. The maximum value depends on the maximum timer period value and the CPU clock speed.

If all the blocks in the model inherit their sample time value, and you do not define the sample time, Simulink assigns a default sample time of 0.2 second.

Using Asynchronous Scheduling

Embedded IDE Link enables you to model and automatically generate code for asynchronous systems. To do so, use the following scheduling blocks:

- Hardware Interrupt (for bare-board code generation mode)
- Idle Task

The Hardware Interrupt block operates by

- Enabling selected hardware interrupts for the processor
- Generating corresponding ISRs for the interrupts
- Connecting the ISRs to the corresponding interrupt service vector table entries

Note You are responsible for mapping and enabling the interrupts you specify in the block dialog box.

Connect the output of the Hardware Interrupt block to the control input of a function-call subsystem. By doing so, you enable the ISRs to call the generated subsystem code each time the hardware raises the interrupt.

The Idle Task block specifies one or more functions to execute as background tasks in the code generated for the model. The functions are created from the function-call subsystems to which the Idle Task block is connected.

Multitasking Scheduler Examples

provides a scheduler that supports multiple tasks running concurrently and preemption between tasks running at the same time. The ability to preempt running tasks enables a wide range of scheduling configurations.

Multitasking scheduling also means that overruns, where a task runs beyond its intended time, can occur during execution.

To understand these examples, you must be familiar with the following scheduling concepts:

- *Preemption* is the ability of one task to pause the processing of a running task to run instead. With the multitasking scheduler, you can define a task as preemptible—thus, another task can pause (preempt) the task that allows preemption. The scheduler examples in this section that demonstrate preemption, illustrate one or more tasks allowing preemption.
- *Overrunning* occurs when a task does not reach completion before it is scheduled to run again. For example, overrunning can occur when a Base-Rate task does not finish in 1 ms. Overrunning delays the next execution of the overrunning task and may delay execution of other tasks.

Examples in this section demonstrate a variety of multitasking configurations:

- “Three Odd-Rate Tasks Without Preemption and Overruns” on page 3-14

- “Two Tasks with the Base-Rate Task Overrunning, No Preemption” on page 3-15
- “Two Tasks with Sub-Rate 1 Overrunning Without Preemption” on page 3-16
- “Three Even-Rate Tasks with Preemption and No Overruns” on page 3-17
- “Three Odd-Rate Tasks Without Preemption and the Base and Sub-Rate1 Tasks Overrun” on page 3-19
- “Three Odd-Rate Tasks with Preemption and Sub-Rate 1 Task Overruns” on page 3-20
- “Three Even-Rate Tasks with Preemption and the Base-Rate and Sub-Rate 1 Tasks Overrun” on page 3-22



Each example presents either two or three tasks:

- **Base Rate task.** Base rate is the highest rate in the model or application. The examples use a base rate of 1ms so that the task should execute every one millisecond.
- **Sub-Rate 1.** The first subrate task. Sub-Rate 1 task runs more slowly than the Base-Rate task. Sub-Rate 1 task rate is 2ms in the examples so that the task should execute every 2ms.
- **Sub-Rate 2.** In examples with three tasks, the second subrate task is called Sub-Rate 2. Sub-Rate 2 tasks run more slowly than Sub-Rate 1. In the examples, Sub-Rate 2 runs at either 4ms or 3ms.
 - When Sub-Rate 2 is 4ms, the example is called *even*.
 - When Sub-Rate 2 is 3ms, the example is called *odd*.

Note The odd or even naming only identifies Sub-Rate 2 as being 3 or 4ms. It does not affect or predict the performance of the tasks.

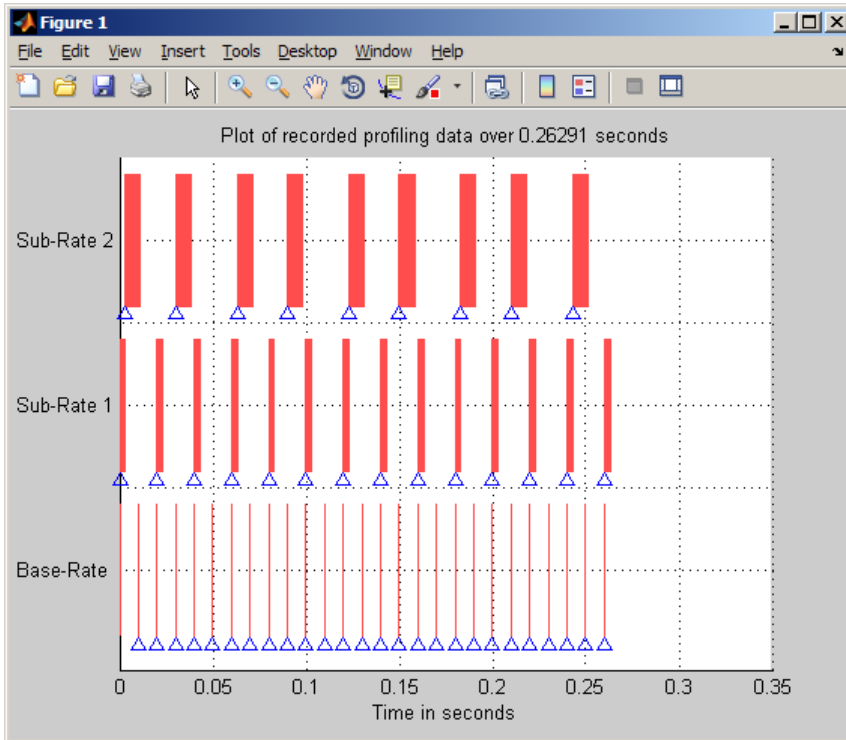
The following legend applies to the plots in the next sections:

- Blue triangles () indicate when the task started.

- Dark red areas () indicate the period during which a task is running
- Pink areas () within dark red areas indicate a period during which a running task is suspended—preempted by a task with higher priority

Three Odd-Rate Tasks Without Preemption and Overruns

In this three task scenario, all of the tasks run as scheduled. No overruns or preemptions occur.

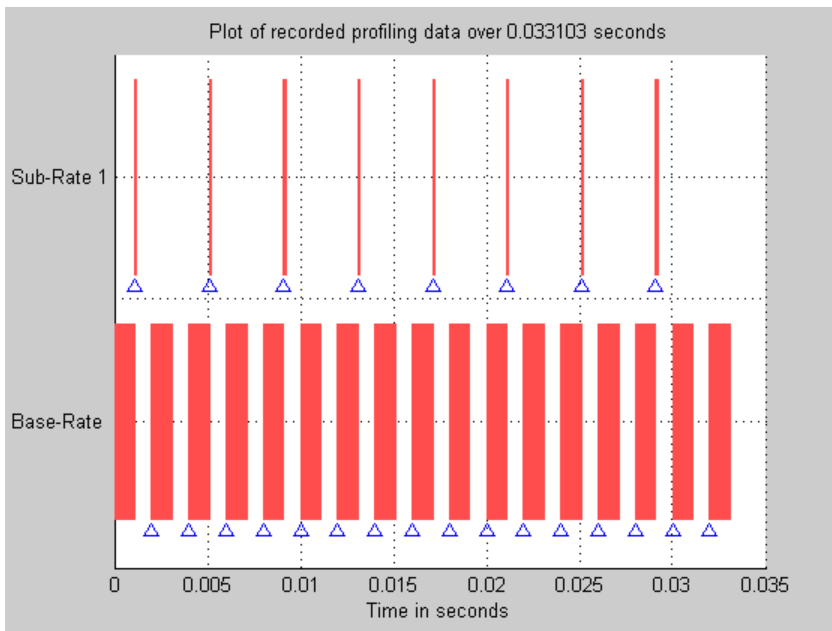


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	1ms
Sub-Rate 1	2ms	2ms
Sub-Rate 2	3ms	3ms

Two Tasks with the Base-Rate Task Overrunning, No Preemption

In this two rate scenario, the Base-Rate overruns the 1ms time intended and prevents the subrate task from completing successfully or running every 2ms.

- Sub-Rate 1 does not allow preemption and fails to run when scheduled, but is never interrupted.
- The Base-Rate runs every 2ms and Sub-Rate 1 runs every 4ms instead of 2ms.

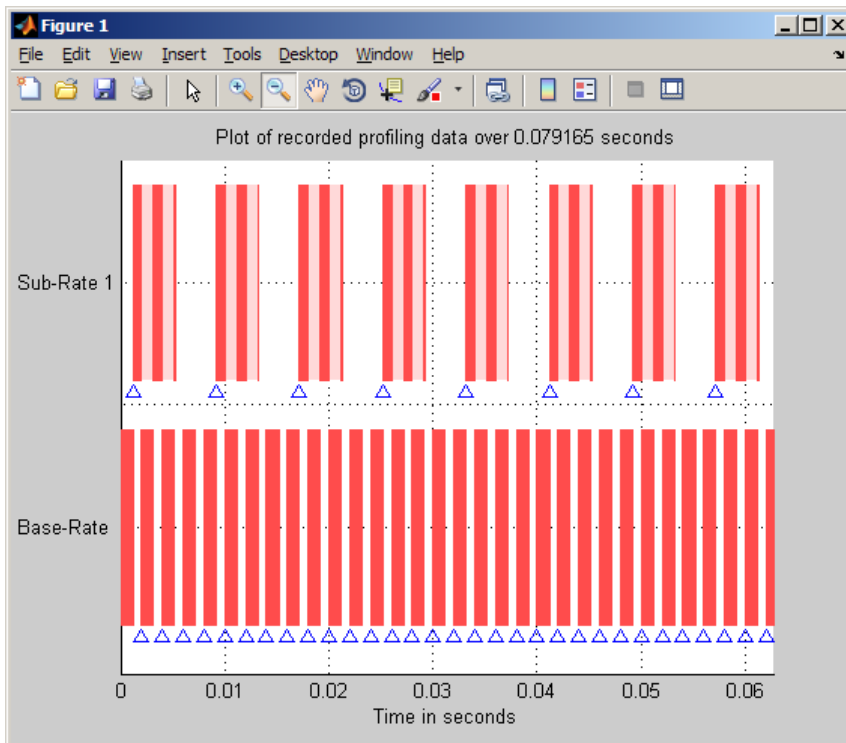


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	4ms (overrunning)

Two Tasks with Sub-Rate 1 Overrunning Without Preemption

In this example, two rates running simultaneously—the Base-Rate task and one subrate task. Both the Base-Rate task and the Sub-Rate 1 task overrun.

- Base-Rate runs every 2ms instead of 1ms.
 - The Sub-Rate 1 task both overruns and is affected by the Base-Rate task overrunning.
 - The Base-Rate task overrun delays Sub-Rate 1 task execution by a factor of 4.
- Sub-Rate 1 runs every 8ms rather than every 2ms.
- The Base-Rate runs at 1ms.
- The Base-Rate task preempts Sub-Rate 1 when it tries to execute.
- The Sub-Rate 1 tasks overrun, taking up to 5ms to complete rather than 2ms.

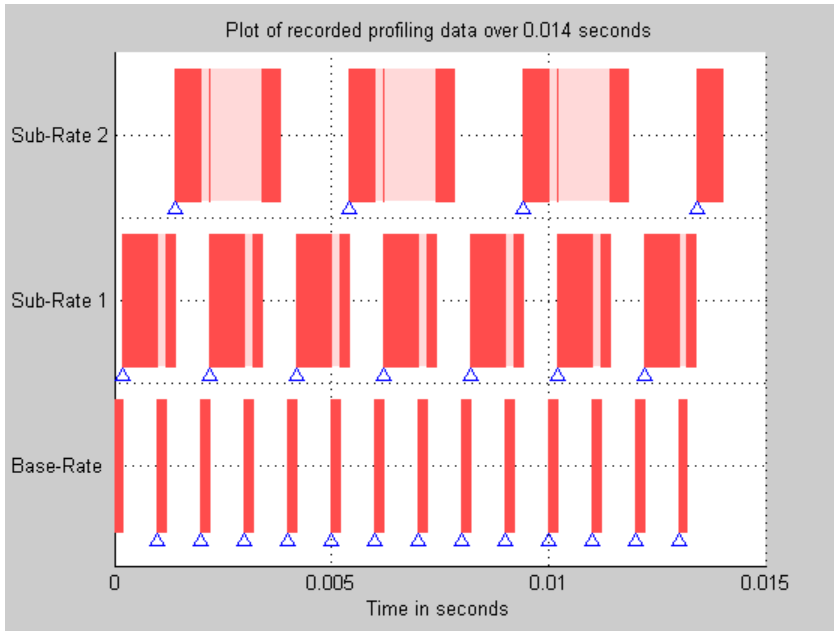


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	8ms (overrunning)

Three Even-Rate Tasks with Preemption and No Overruns

In the following three task scenario, the Base-Rate runs as scheduled and preempts Sub-Rate 1.

- Both the Base-Rate and Sub-Rate 1 tasks preempt Sub-Rate 2 task execution.
- Preempting the subrate tasks does not prevent the subrate tasks from running on schedule.

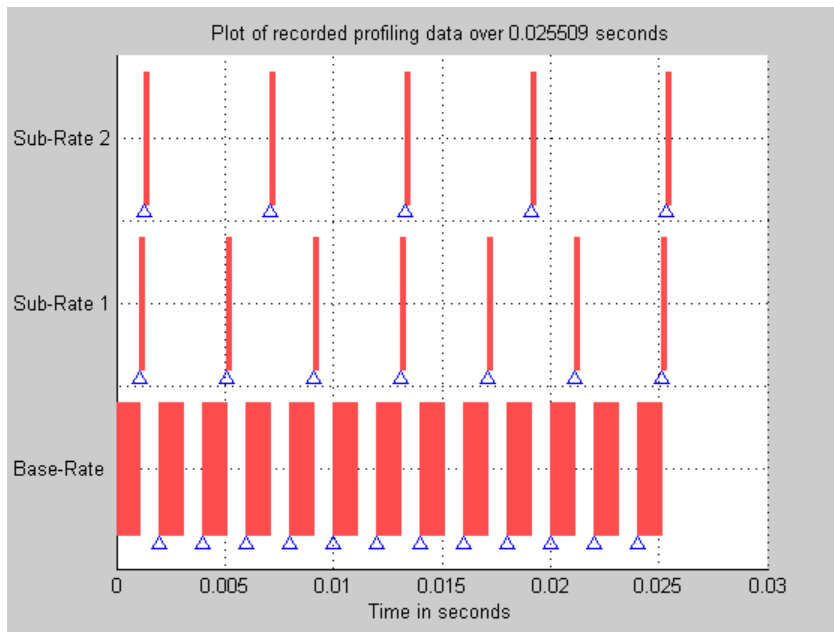


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	1ms
Sub-Rate 1	2ms	2ms
Sub-Rate 2	4ms	4ms

Three Odd-Rate Tasks Without Preemption and the Base and Sub-Rate1 Tasks Overrun

Three tasks running simultaneously—the Base-Rate task and two subrate tasks.

- Both the Base-Rate task and the Sub-Rate 1 task overrun.
- The Base-Rate task runs every 2ms instead of 1ms.
- Sub-Rate 1 and Sub-Rate 2 task execution is delayed by a factor of 2—Sub-Rate 1 runs every 4ms rather than every 2ms and Sub-Rate 2 runs every 6ms instead of 3ms.

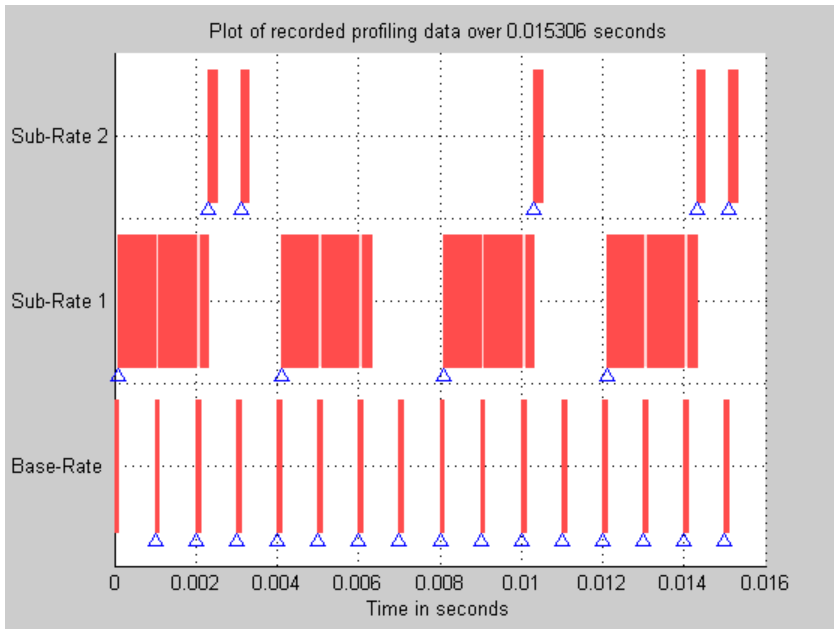


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	4ms (overrunning)
Sub-Rate 2	3ms	6ms (overrunning)

Three Odd-Rate Tasks with Preemption and Sub-Rate 1 Task Overruns

In this three task scenario, the Base-Rate preempts Sub-Rate 1 which is overrunning.

- The overrunning subrate causes Sub-Rate 1 to execute every 4ms instead of 2ms.
- Every other fourth execution of Sub-Rate 2 does not occur.
- Instead of executing at $t=0, 3, 6, 9, 12, 15, 18, \dots$, Sub-Rate 2 executes at $t=0, 3, 9, 12, 15, 21$, and so on.
- The $t=6$ and $t=18$ instances do not occur.



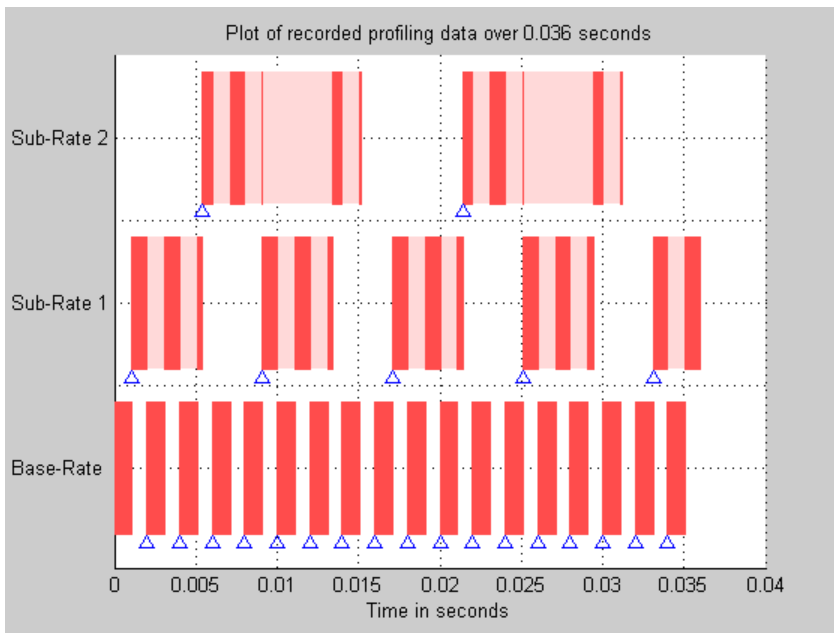
Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)

Task Identification	Intended Execution Schedule	Actual Execution Schedule
Sub-Rate 1	2ms	4ms (overrunning)
Sub-Rate 2	3ms	6ms (overrunning and skipping every other fourth execution)

Three Even-Rate Tasks with Preemption and the Base-Rate and Sub-Rate 1 Tasks Overrun

In this three-task scenario, two of the tasks overrun—the Base-Rate and Sub-Rate 1.

- The overrunning Base-Rate executes every 2ms.
- Sub-Rate 1 overruns due to the Base-Rate overrun, doubling the execution rate.
- Also, Sub-Rate 1 is overrunning as well, doubling the execution rate again, from the intended 2ms to 8ms.
- Sub-Rate 2 responds to the overrunning Base-Rate and Sub-Rate 1 tasks by running every 16ms instead of every 4ms.



Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)

Task Identification	Intended Execution Schedule	Actual Execution Schedule
Sub-Rate 1	2ms	8ms (overrunning)
Sub-Rate 2	3ms	16ms (overrunning)

Project Generator Tutorial

In this section...
“Creating the Model” on page 3-25
“Adding the Target Preferences Block to Your Model” on page 3-25
“Specify Configuration Parameters for Your Model” on page 3-29

In this tutorial you will use the Embedded IDE Link software to:

- Build a model.
- Generate a project from the model.
- Build the project and run the binary on a processor.

Note The model demonstrates project generation. You cannot not build and run the model on your processor without additional blocks.

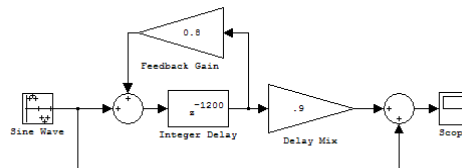
To generate a project from a model, complete the following tasks:

- 1** Create a model application.
- 2** Add a Target Preferences block from the Embedded IDE Link library to your model.
- 3** In the Target Preferences block, verify and set the block parameters for your hardware or simulator.
- 4** Set the configuration parameters for your model, including
 - Solver parameters such as simulation start and solver options
 - Real-Time Workshop software options such as processor configuration and processor compiler selection
- 5** Generate your project.
- 6** Review your project in CCS.

Creating the Model

To create the model for audio reverberation, follow these steps:

- 1 Start Simulink software.
- 2 Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.
- 3 Use Simulink blocks and Signal Processing Blockset™ blocks to create the following model.



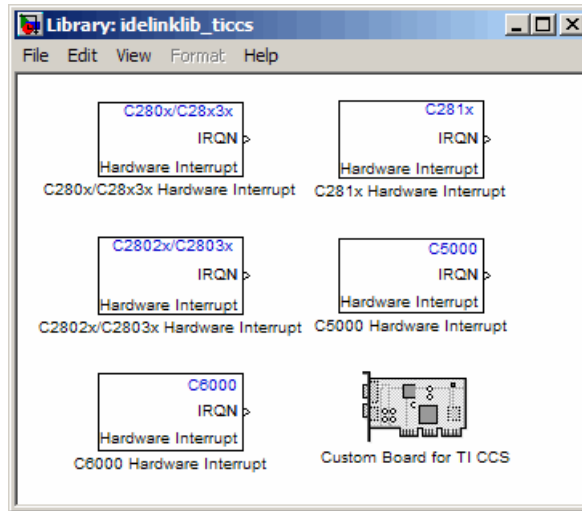
Look for the Integer Delay block in the Discrete library of Simulink blocks and the Gain block in the Commonly Used Blocks library. Do not add the Custom Board for TI CCS block at this time.

- 4 Save your model with a suitable name before continuing.

Adding the Target Preferences Block to Your Model

So that you can configure your model to work with TI processors, Embedded IDE Link supplies a Target Preferences/Custom Board block for Texas Instruments processors.

Entering `idelinklib_ticcs` at the MATLAB software prompt opens the block library. This block library is included in Embedded IDE Link `idelinklib` blockset in the Simulink Library browser.



Adding a Target Preferences block to a model triggers a dialog box that asks about your model configuration settings. The message tells you that the model configuration parameters will be set to default values based on the processor specified in the block parameters. To set the parameters automatically, click **Yes**. Clicking **No** dismisses the dialog box and does not set the parameters.

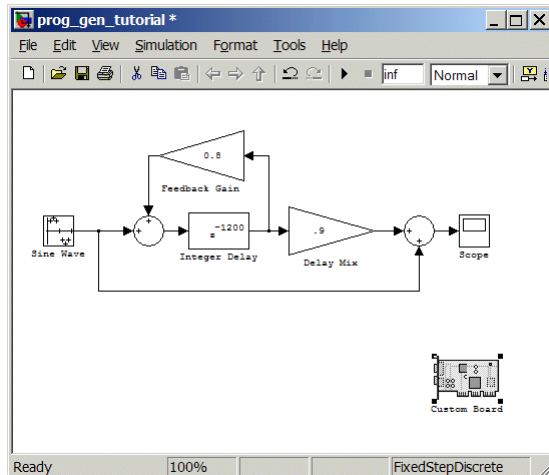
When you click **Yes**, the software sets the system target file to `ccslink_grt.tlc` or `ccslink_ert.tlc` and sets the hardware options and product-specific parameters in the model to default values. If you open the model Configuration Parameters, you see the Embedded IDE Link pane option on the select tree.

Clicking **No** prevents the software from setting the system target file and the product specific options. When you open the model Configuration Parameters for your model, you do not see the Embedded IDE Link pane option on the select tree. To enable the options, select the `ccslink_ert.tlc` or `ccslink_grt.tlc` system target file from the System Target File list in the Real-Time Workshop pane options.

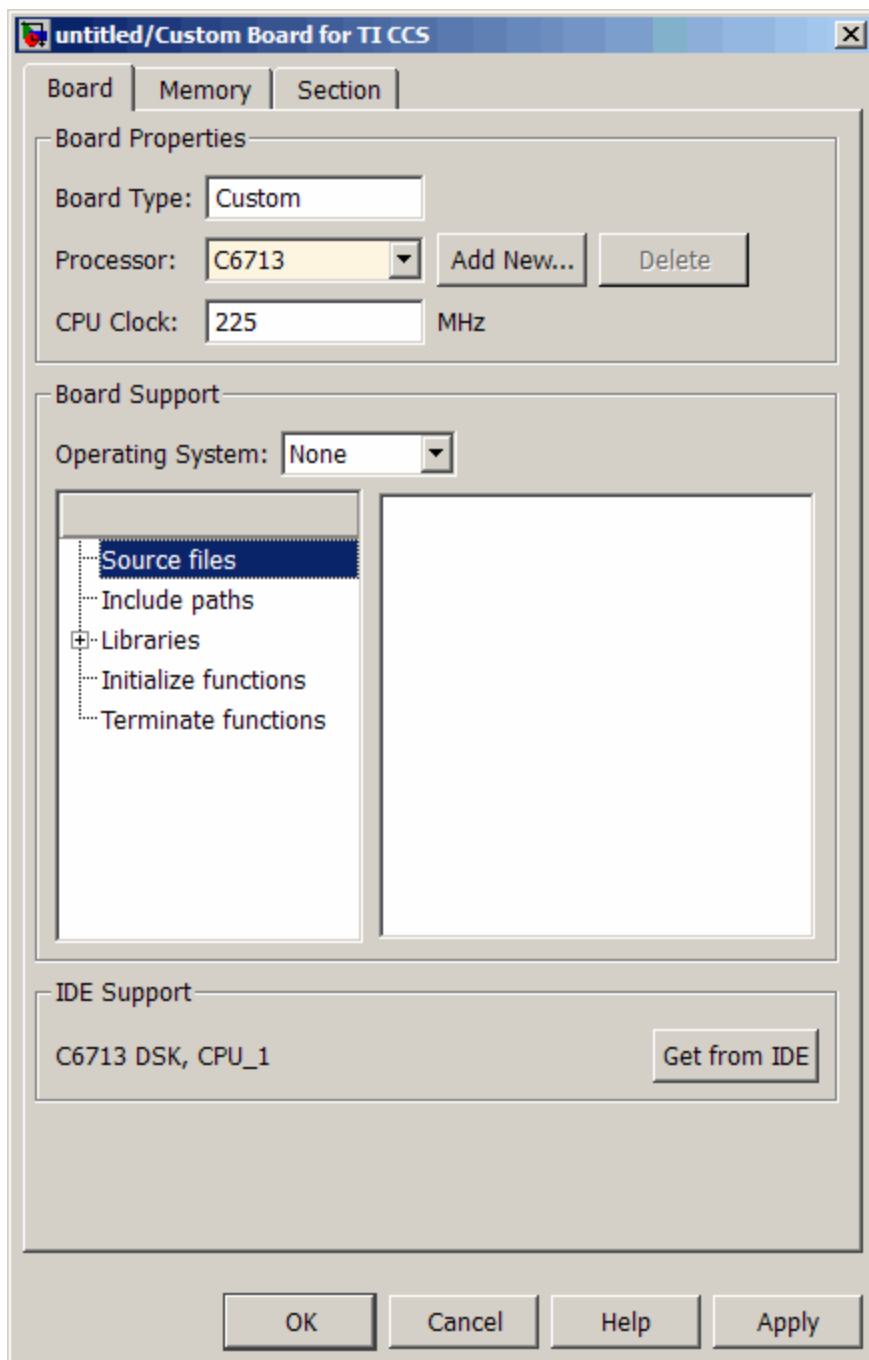
To add the Target Preferences block to your model, follow these steps:

- 1 Double-click Embedded IDE Link in the Simulink Library browser to open the `idelinklib` blockset.

- 2 Select **Supported IDEs > Texas Instruments Code Composer Studio** block library.
- 3 Drag and drop the Custom Board for TI CCS block to your model as shown in the following model window figure.



- 4 Double-click the Custom Board for TI CCS block in the model to open the block dialog box.



- 5 In the Block dialog box, select your processor from the **Processor** list.
- 6 Verify the **CPU clock** value and, if you are using a simulator, select **Simulator**.
- 7 Verify the settings on the **Memory** and **Sections** tabs to be sure they are correct for the processor you selected.
- 8 Click **OK** to close the Target Preferences dialog box.

You have completed the model. Now configure the model configuration parameters to generate a project in CCS IDE from your model.

Specify Configuration Parameters for Your Model

The following sections describe how to configure the build and run parameters for your model. Generating a project, or building and running a model on the processor, starts with configuring model options in the Configuration Parameters dialog box in Simulink software.

Setting Solver Parameters

After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the configuration parameters for the model:

- 1 Open the Configuration Parameters dialog box and set the appropriate options on the **Solver** category for your model and for Embedded IDE Link.
 - Set **Start time** to 0.0 and **Stop time** to `inf` (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this to `inf` for completeness.
 - Under **Solver options**, select the fixed-step and discrete settings from the lists
 - Set the **Fixed step size** to Auto and the **Tasking Mode** to Single Tasking

Note Generated code does not honor Simulink software stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, add a Stop Simulation block in your model.

When you use PIL, you can set the **Solver options** to any selection from the **Type** and **Solver** lists.

Ignore the Data Import/Export, Diagnostics, and Optimization categories in the Configuration Parameters dialog box. The default settings are correct for your new model.

Setting Real-Time Workshop Code Generation Parameters

To configure Real-Time Workshop software to use the correct processor files and to compile and run your model executable file, set the options in the Real-Time Workshop category of the **Select** tree in the Configuration Parameters dialog box. Follow these steps to set the code generation options for your DSP:

- 1 Select Real-Time Workshop on the **Select** tree.
- 2 In Target selection, use the **Browse** button to set **System target file** to `ccslink_grt.tlc`.

Setting Embedded IDE Link Parameters

To configure Real-Time Workshop software to use the correct code generation options and to compile and run your model executable file, set the options in the Embedded IDE Link category of the **Select** tree in the Configuration Parameters dialog box. Follow these steps to set the code generation options for your processor:

- 1 From the **Select** tree, choose Embedded IDE Link to specify code generation options that apply to your processor.
- 2 Set the following options in the pane under **Project options**:
 - **Project options** should be Custom.

- Set **Compiler options string** and **Linker options string** should be blank.
- 3** Under **Link Automation**, verify that **Export IDE link handle to base workspace** is selected and provide a name for the handle in **IDE handle name** (optional).
 - 4** Set the following **Runtime** options:
 - **Build action:** `Build_and_execute`.
 - **Interrupt overrun notification method:** `None`.

You have configured the Real-Time Workshop software options that let you generate a project for you processor. You may have noticed that you did not configure a few categories on the **Select** tree, such as **Comments**, **Symbols**, and **Optimization**.

For your new model, the default values for the options in these categories are correct. For other models you develop, you may want to set the options in these categories to provide information during the build and to run TLC debugging when you generate code. Refer to your Simulink and Real-Time Workshop documentation for more information about setting the configuration parameters.

Building Your Project

After you set the configuration parameters and configure Real-Time Workshop software to create the files you need, you direct the build process to create your project:

- 1** Press **OK** to close the Configuration Parameters dialog box.
- 2** Click **Ctrl+B** to generate your project into CCS IDE.

When you click **Build** with `Create_project` selected for **Build action**, the automatic build process starts CCS IDE, populates a new project in the development environment, builds the project, loads the binary on the processor, and runs it.

- 3 To stop processor execution, use the **Halt** option in CCS or enter `cc.halt` at the MATLAB command prompt. (Where “cc” is the IDE handle name you specified previously in **Configuration Parameters**.)

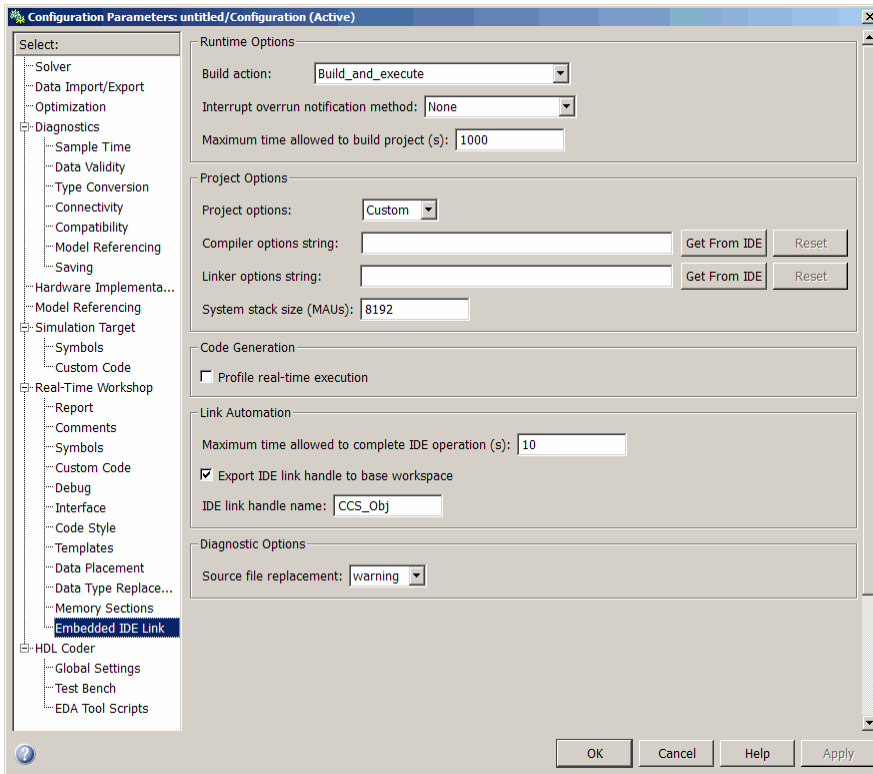
Setting Code Generation Parameters for TI Processors

Before you generate code with Real-Time Workshop software, set the fixed-step solver step size and specify an appropriate fixed-step solver if the model contains any continuous-time states. At this time, you should also select an appropriate sample rate for your system. Refer to your *Real-Time Workshop User's Guide* documentation for additional information.

Note Embedded IDE Link does not support continuous states in Simulink software models for code generation. In the **Solver options** in the Configuration Parameters dialog box, you must select **Discrete (no continuous states)** as the **Type**, along with **Fixed step**.

The Real-Time Workshop pane of the Configuration Parameters dialog box lets you set numerous options for the real-time model. To open the Configuration Parameters dialog box, select **Simulation > Configuration Parameters** from the menu bar in your model.

The following figure shows the configuration parameters categories when you are using Embedded IDE Link.



In the **Select** tree, the categories provide access to the options you use to control how Real-Time Workshop software builds and runs your model. The first categories under **Real-Time Workshop** in the tree apply to all Real-Time Workshop software processors. They always appear on the list.

The last category under **Real-Time Workshop** is specific to the Embedded IDE Link system target files `ccslink_grt.tlc` and `ccslink_ert.tlc` and appear when you select either file.

When you select your processor file in **Target Selection** on the **Real-Time Workshop** pane, the options change in the tree.

For Embedded IDE Link, the processor to select is `ccslink_grt.tlc`. Selecting either the `ccslink_grt.tlc` or `ccslink_ert.tlc` adds the Embedded IDE Link options to the **Select** tree. The `ccslink_grt.tlc` file is

appropriate for all projects. Select `ccslink_ert.tlc` when you are developing projects or code for embedded processors (requires Real-Time Workshop Embedded Coder software) or you plan to use Processor-in-the-Loop features.

The following sections present each configuration parameters **Select** tree category and the relevant options available in each.

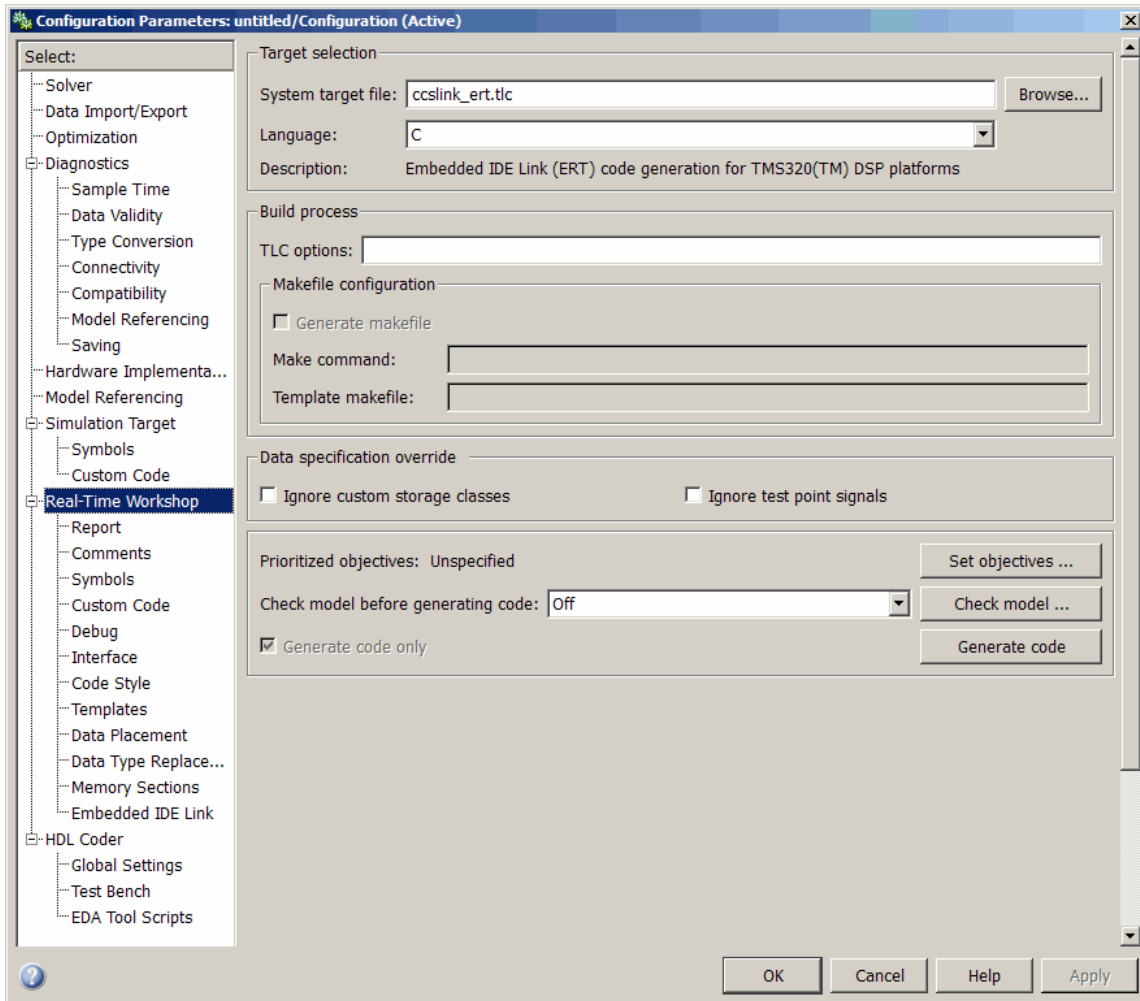
Setting Model Configuration Parameters

In this section...
“Target File Selection” on page 3-37
“Build Process” on page 3-38
“Custom Storage Class” on page 3-38
“Report Options” on page 3-38
“Debug Pane Parameters” on page 3-39
“Optimization Pane Parameters” on page 3-40
“Embedded IDE Link Pane Parameters” on page 3-42
“Default Project Configuration — Custom” on page 3-47

Use the options in the **Select** tree under **Real-Time Workshop** to perform the following configuration tasks.

- Select your processor file.
- Configure your build process.
- Specify whether to use custom storage classes.

Selecting the system target (`ccslink_grt.tlc` or `ccslink_ert.tlc`) in **System target file** enables Embedded IDE Link configuration options in the Embedded IDE Link pane.



Target File Selection

System target file

Clicking **Browse** opens the processor File Browser where you select `ccslink_grt.tlc` as your Real-Time Workshop **System target file** for Embedded IDE Link.

If you are using Real-Time Workshop Embedded Coder software or plan to use PIL, select the `ccslink_ert.tlc` processor in **System target file**.

Build Process

Embedded IDE Link software does not use makefiles or the build process to generate code. Code generation is project based so the options in this group do not apply.

Custom Storage Class

When you generate code from a model employing custom storage classes (CSC), make sure to clear **Ignore custom storage classes**. This setting is the default value for Embedded IDE Link and for Real-Time Workshop Embedded Coder.

When you select **Ignore custom storage classes**,

- Objects with CSCs are treated as if you set their storage class attribute to `Auto`.
- The storage class of signals that have CSCs does not appear on the signal line, even when you select `Storage class` from **Format > Port/Signals Display** in your Simulink menus.

Ignore custom storage classes lets you switch to a processor that does not support CSCs, such as the generic real-time processor (GRT), without having to reconfigure your parameter and signal objects.

Generate code only

The **Generate code only** option does not apply to generating code with Embedded IDE Link. To generate source code without building and executing the code on your processor, select `Embedded IDE Link` from the **Select** tree. Then, under **Runtime**, select `Create project for Build action`. You cannot use DSP/BIOS features when you use the `Create project` option for the **Build action**.

Report Options

Two options control HTML report generation during code generation.

- “Create Code Generation report” on page 3-39
- “Launch report automatically” on page 3-39

Create Code Generation report

After you generate code, this option tells the software whether to generate an HTML report that documents the C code generated from your model. When you select this option, Real-Time Workshop writes the code generation report files in the `html` subdirectory of the build directory. The top-level HTML report file is named `modelName_codegen_rpt.html` or `subsystemname_codegen_rpt.html`. For more information about the report, refer to the online help for Real-Time Workshop. You can also use the following command at the MATLAB prompt to get more information.

```
docsearch 'Create code generation report'
```

In the Navigation options, when you select **Model-to-code** and **Code-to-model**, your HTML report includes hyperlinks to various features in your Simulink model.

Launch report automatically

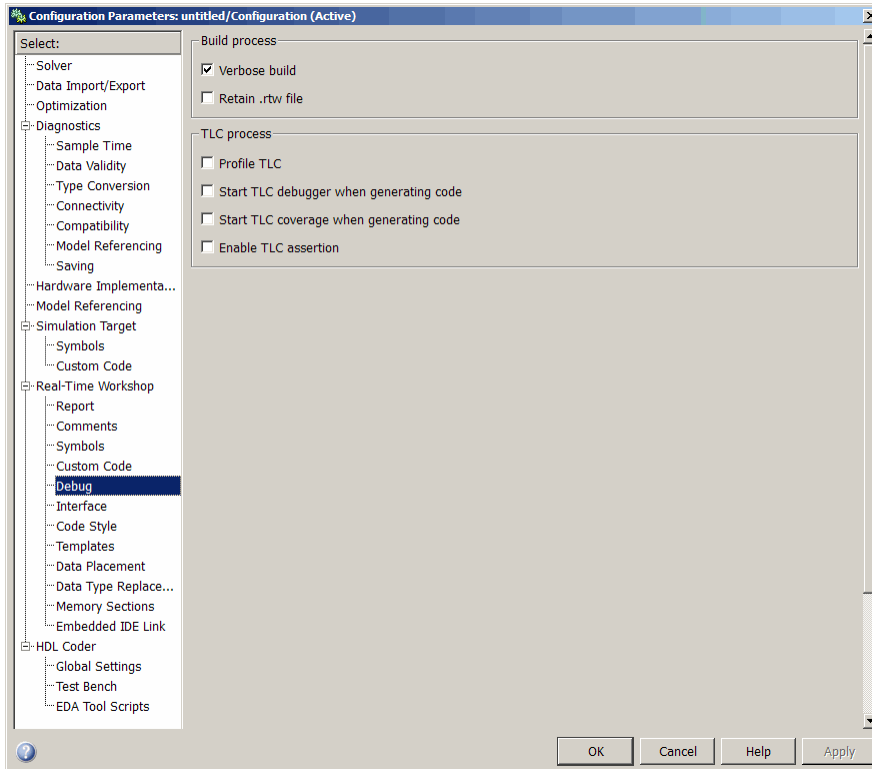
This option directs Real-Time Workshop to open a MATLAB Web browser window and display the code generation report. If you clear this option, you can open the code generation report (`modelName_codegen_rpt.html` or `subsystemname_codegen_rpt.html`) manually in a MATLAB Web browser window or in another Web browser.

Debug Pane Parameters

Real-Time Workshop uses the processor Language Compiler (TLC) to generate C code from the `model.rtw` file. The TLC debugger helps you identify programming errors in your TLC code. Using the debugger, you can

- View the TLC call stack.
- Execute TLC code line-by-line and analyze and/or change variables in a specified block scope.

When you select **Debug** from the **Select** tree, you see the **Debug** options as shown in the next figure. In this dialog box, you set options that are specific to Real-Time Workshop process and TLC debugging.

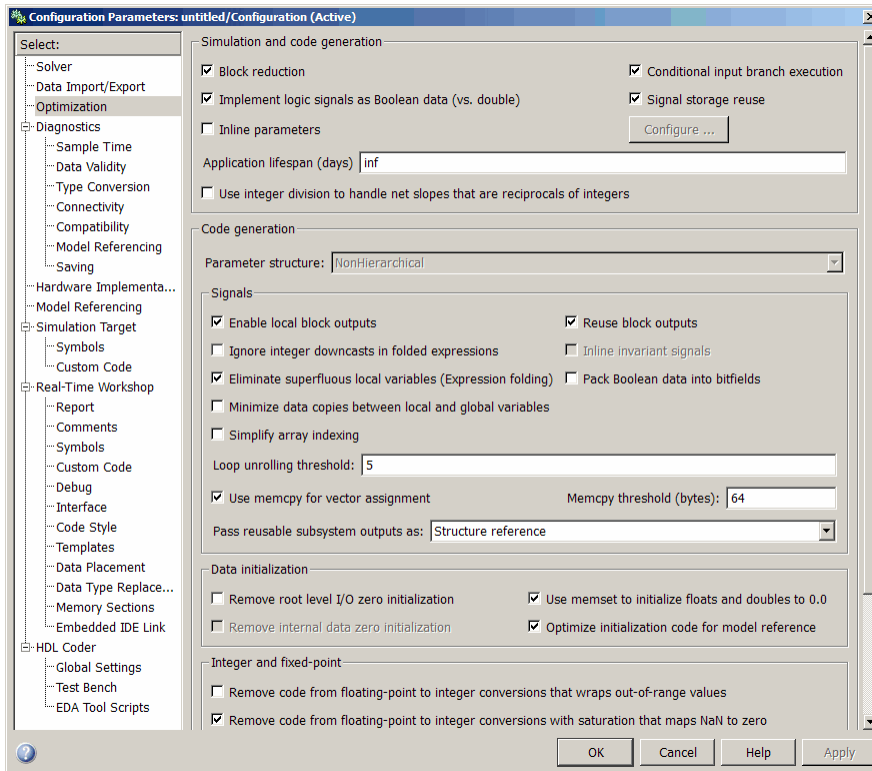


For details about using the options in **Debug**, refer to “About the TLC Debugger” in your Real-Time Workshop processor Language Compiler documentation.

Optimization Pane Parameters

On the **Optimization** pane in the Configuration Parameters dialog box, you set options for the code that Real-Time Workshop generates during the build process. You use these options to tailor the generated code to your needs. Select **Optimization** from the **Select** tree on the Configuration Parameters dialog box. The figure shows the **Optimization** pane when you select the

system target file `ccslink_grt.tlc` under **Real-Time Workshop system target file**.



These are the options typically selected for Real-Time Workshop:

- **Conditional input branch execution**
- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse block outputs**
- **Eliminate superfluous local variables (Expression folding)**
- **Loop unrolling threshold**
- **Optimize initialization code for model reference**

For more information about using these and the other **Optimization** options, refer to your Real-Time Workshop documentation.

Embedded IDE Link Pane Parameters

On the select tree, the Embedded IDE Link entry provides options in these areas:

- **Runtime** — Set options for run-time operations, like the build action
- **Project Options** — Set build options for your project code generation
- **Code Generation** — Configure your code generation requirements
- **Link Automation** — Export a `ticc` object to your MATLAB workspace
- **Diagnostic options** — Determine how the code generation process responds when you use source code replacement, either in the Target Preferences block **Board custom code** options, or in the Real-Time Workshop **Custom Code** options in the configuration parameters.

Runtime Options

Before you are able to an executable to run on any Texas Instruments processor, you must configure the run-time options for the source model.

By selecting values for the options available, you configure the operation of your processor.

Build action

To specify to Real-Time Workshop software what to do when you click **Build**, select one of the following options. The actions are cumulative—each listed action adds features to the previous action on the list and includes all the previous features:

- **Generate_code_only** — Directs Real-Time Workshop software to generate ANSI C code only from the model. It does not use the Texas Instruments software tools, such as the compiler and linker, and you do not need to have CCS installed. Also, MATLAB software does not create the connection to CCS that results from the other options. This option does not build code for

TI processors. You cannot use this option when you set the system target file to either `ccslink_grt.tlc` or `ccslink_ert.tlc`.

The build process for a model also generates the files `modelname.c`, `modelname.cmd`, `modelname.bld`, and many others. It puts the files in a build directory named `modelname_linkforccs_rtw` in your MATLAB working directory. This file set contains many of the same files that Real-Time Workshop software generates to populate a CCS project when you choose `Create_Project` for the build action.

- `Create_Project` — Directs Real-Time Workshop software to start CCS and populate a new project with the files from the build process. This option offers a convenient way to build projects in CCS.
- `Archive_library` — Directs Real-Time Workshop software to archive the project for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your CCS projects for models that you use in model referencing.
- `Build` — Builds the executable COFF file, but does not download the file to the processor.
- `Build_and_execute` — Directs Real-Time Workshop software to build, download, and run your generated code as an executable on your processor.
- `Create_processor_in_the_loop_project` — Directs the Real-Time Workshop code generation process to create PIL algorithm object code as part of the project build.

Your selection for **Build action** determines what happens when you click **Build** or press **Ctrl+B**. Your selection tells Real-Time Workshop software when to stop the code generation and build process.

To run your model on the processor, select `Build_and_execute`. This selection is the default build action; Real-Time Workshop software automatically downloads and runs the model on your board.

Note When you build and execute a model on your processor, the Real-Time Workshop software build process resets the processor automatically. You do not need to reset the board before building models.

Interrupt overrun notification method

To enable the overrun indicator, choose one of three ways for the processor to respond to an overrun condition in your model:

- **None** — Ignore overruns encountered while running the model.
- **Print_message** — When the DSP encounters an overrun condition, it prints a message to the standard output device, `stdout`.
- **Call_custom_function** — Respond to overrun conditions by calling the custom function you identify in **Interrupt overrun notification function**.

Interrupt overrun notification function

When you select `Call_custom_function` from the **Interrupt overrun notification method** list, you enable this option. Enter the name of the function the processor should use to notify you that an overrun condition occurred. The function must exist in your code on the processor.

Project Options

Before you run your model as an executable on any processor, you must configure the Project options for the model. The default setting is `Custom`, which does not use any optimization flags.

Compiler options string

To determine the degree of optimization provided by the TI optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your CCS documentation. When you create new projects, Embedded IDE Link does not set any optimization flags.

Click **Get From IDE** to import the compiler option setting from the current project in the IDE. To reset the compiler option to the default value, click **Reset**.

Linker options string

To specify the options provided by the TI linker during link time, you enter the linker options as a string. For details about the linker options, refer to

your CCS documentation. When you create new projects, Embedded IDE Link does not set any linker options.

Click **Get From IDE** to import the linker options string from the current project in the IDE. To reset the linker options to the default value of no options, click **Reset**.

System stack size (MAUs)

Enter the amount of memory to use for the stack. For more information, refer to **Enable local block outputs** on the **Optimization** pane of the Configuration Parameters dialog box. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory. Also refer to the online Help system for more information about Real-Time Workshop options for configuring and building models and generating code.

Code Generation

From this category, you select options that define the way your code is generated:

- **Profile real-time execution**

To enable the real-time execution profile capability, select **Profile real-time execution**. With this selected, the build process instruments your code to provide performance profiling at the task level or for atomic subsystems. When you run your code, the executed code reports the profiling information in an HTML report.

Link Automation

When you use Real-Time Workshop to build a model to a C6000 processor, Embedded IDE Link makes a connection between MATLAB software and CCS.

If you have used Embedded IDE Link, you are familiar with function `ticcs`, which creates objects the reference between the IDE and MATLAB. This option refers to the same object, called `cc` in the function reference pages.

Although MATLAB to CCS is a bridge to a specific instance of the CCS IDE, it is an object that contains information about the IDE instance it refers to, such as the board and processor it accesses. In this pane, the **Export IDE link handle to base workspace** option lets you instruct Embedded IDE Link to export the object to your MATLAB workspace, giving it the name you assign in **IDE link handle name**.

Maximum time to complete IDE operations (s)

Specifies how long the software waits for IDE functions, such as `read` or `write`, to return completion messages.

Diagnostic Option

Source file replacement selects the diagnostic action to take if the software detects conflicts when you replace source code with custom code. The diagnostic message responds to both source file replacement in the Embedded IDE Link parameters and in the Real-Time Workshop **Custom code** parameters in the configuration parameters for your model.

The following settings define the messages you see and how the code generation process responds:

- `none` — Does not generate warnings or errors when it finds conflicts.
- `warning` — Displays a warning. `warn` is the default value.
- `error` — Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

The build operation continues if you select `warning` and the software detects custom code replacement problems. You see warning messages as the build progresses

Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files. Use `none` when the replacement process is correct and you do not want to see multiple messages during your build.

Default Project Configuration – Custom

Although CCS offers two standard project configurations, **Release** and **Debug**, models you build with Embedded IDE Link use a custom configuration that provides a third combination of build and optimization settings—**Custom**.

Project configurations define sets of project build options. When you specify the build options at the project level, the options apply to all files in your project. For more information about the build options, refer to your TI CCS documentation.

The default settings for **Custom** are the same as the **Release** project configuration in CCS, except for the compiler options discussed in the next section.

Compiler Options in Custom Project Configuration

When you create a new project or build a project from a model, your project and model inherit the build configuration settings from the configuration **Custom**. The compiler settings in **Custom** differ from the settings in the default **Release** and **Debug** configurations in CCS.

For the compiler settings, **Custom** does not use any options, to preserve important features of the generated code. The **Release** configuration uses the **Release** build optimizations set by CCS, with the addition of the `-o2` optimization flag. The **Debug** configuration uses the CCS default debugging options and adds the `-g`, `-d`, and `_DEBUG` flags.

For memory configuration, where **Release** uses the default memory model that specifies near functions and data, **Custom** specifies far functions and data. Your CCS documentation provides complete details on the compiler build options.

You can change the individual settings or the build configuration within CCS. Build configuration options that do not appear on these panes default to match the settings for the **Release** build configuration in CCS.

Using Custom Source Files in Generated Projects

In this section...

“Preparing to Replace Generated Files With Custom Files” on page 3-48

“Replacing Generated Source Files with Custom Files When You Generate Code” on page 3-50

The **Board custom code** options on the **Board Info** pane in the model’s Target Preferences block enable you to replace a generated file with a custom file you provide. By replacing a file, you can modify the output of the code generation process to suit your needs. For file replacement during the code generation process to work, your custom file must have the same name as the file to replace.

The following sections show you how to:

- Identify the file to replace — “Preparing to Replace Generated Files With Custom Files” on page 3-48
- Create the custom replacement file — “Preparing to Replace Generated Files With Custom Files” on page 3-48
- Configure the target preferences to use the custom file when you generate a project — “Replacing Generated Source Files with Custom Files When You Generate Code” on page 3-50

For more information about the target preferences and the Board custom code options, refer to Target Preferences/Custom Board in the online Help system.

Preparing to Replace Generated Files With Custom Files

To change the content of a generated project, use custom code replacement to replace a generated file in the project. By replacing a generated file in a project, you can make changes like the following in your generated project:

- Edit the file that contains linker directives to change the linking process, such as mapping memory differently.

- Modify a library file, such as a chip support library (.cs1 file)
- Add commands to a header file
- Modify data in a data file
- Add comments to a file for tracking or identifying the file or project

Determining the Name of the File to Replace

To replace a file created when you generate a project, you need the name of the file to replace; the content to change; and the replacement file, including the path to the file. Your model must include a target preferences block configured for your processor, either a simulator or hardware.

Follow these steps to identify the file to replace in a project:

- 1** Open the configuration parameters for your model.
- 2** On the **Select** tree in the Configuration Parameter dialog box, select **Embedded IDE Link**.
- 3** Set **Build action** to any entry on the list except **Create processor-in-the-loop project**.
- 4** Click **OK** to close the dialog box.
- 5** Press **Ctrl+B** to build your model.
- 6** Look at the files in the project in the IDE. Find the file that contains the information to supplement or replace.
- 7** Note the file name and location. You use this information to create your custom replacement file.

Creating the Replacement File

To replace a file in a project during code generation, you need a new file with the same name saved in a different directory. Creating your replacement file from the file to replace increases the chances that the generated code will

work properly with the new file. The new file must have all of the information the final project needs.

Follow these steps to create a file to use to replace a generated file in your project.

- 1** Determine the name of the file to replace. Refer to “Determining the Name of the File to Replace” on page 3-49 for how to do this.
- 2** Locate the file to replace. Copy the file and save it with the same name in a new directory.
- 3** Open your new file and edit the file to add or remove the information to change.
- 4** Save your changes to the file.

Replacing Generated Source Files with Custom Files When You Generate Code

With the replacement file and location available, configure the build process to use your replacement file. Parameters on the Target Preferences block dialog box allow you to specify the replacement file to use. For more information about the board custom code options, refer to Target Preferences/Custom Board

Follow these steps to configure the build process to use a replacement file.

- 1** Double-click Target Preferences in your model. Doing this opens a block dialog box similar to the one in the following figure.
- 2** In the Board custom code options, select the type of file to replace—Source files or Libraries.
- 3** Enter the name of your replacement file and path in the text field.

The build process recognizes two directory path tokens:

- `$MATLAB` to refer to your MATLAB root directory
- `$install_dir` to refer to the root of your IDE installation.

- 4** Click **OK** to close the dialog box.
- 5** Open the configuration parameters for your model and select the **Build action** to use to build your model.
- 6** From the **Source code replacement** list, select **warning** or **error** to see messages when the build process replaces files.
- 7** Click **OK** to save your configuration.
- 8** Return to the model window and press **Ctrl+B** to build your project. The generated project contains your replacement file instead of generating the matching file.

Optimizing Embedded Code with Target Function Libraries

In this section...

“About Target Function Libraries and Optimization” on page 3-52

“Using a Processor-Specific Target Function Library to Optimize Code” on page 3-54

“Process of Determining Optimization Effects Using Real-Time Profiling Capability” on page 3-55

“Reviewing Processor-Specific Target Function Library Changes in Generated Code” on page 3-56

“Reviewing Target Function Library Operators and Functions” on page 3-58

“Creating Your Own Target Function Library” on page 3-58

About Target Function Libraries and Optimization

A *target function library* is a set of one or more function tables that define processor- and compiler-specific implementations of functions and arithmetic operators. The code generation process uses these tables when it generates code from your Simulink model.

The software registers processor-specific target function libraries during installation. To use one of the libraries, select the set of tables that correspond to functions implemented by intrinsics or assembly code for your processor from the **Target function library** list in the model configuration parameters. To do this, complete the following steps:

- 1 In your model, select **Simulation > Configuration Parameters**.
- 2 In the Configuration Parameters dialog box, select **Real-Time Workshop** and **Interface**.
- 3 Set the **Target function library** parameter to the appropriate library for your processor.

After you select the processor-specific library, the model build process uses the library contents to optimize generated code for that processor. The generated code includes processor-specific implementations for sum, sub, mult, and div,

and various functions, such as `tan` or `abs`, instead of the default ANSI[®] C instructions and functions. The optimized code enables your embedded application to run more efficiently and quickly, and in many cases, reduces the size of the code. For more information about target function libraries, refer to “Introduction to Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation.

Code Generation Using the Target Function Library

The build process begins by converting your model and its configuration set to an intermediate form that reflects the blocks and configurations in the model. Then the code generation phase starts.

Note Real-Time Workshop refers to the following conversion process as replacement and it occurs before the build process generates a project.

During code generation for your model, the following process occurs:

- 1** Code generation encounters a call site for a function or arithmetic operator and creates and partially populates a target function library entry object.
- 2** The entry object queries the target function library database for an equivalent math function or operator. The information provided by the code generation process for the entry object includes the function or operator key, and the conceptual argument list.
- 3** The code generation process passes the target function library entry object to the target function library.
- 4** If there is a matching table entry in the target function library, the query returns a fully-populated target function library entry to the call site, including the implementation function name, argument list, and build information
- 5** The code generation process uses the returned information to generate code.

Within the target function library that you select for your model, the software searches the tables that comprise the library. The search occurs in the order in which the tables appear in either the Target Function Library Viewer or

the **Target function library** tool tip. For each table searched, if the search finds multiple matches for a target function library entry object, priority level determines the match to return. The search returns the higher-priority (lower-numbered) entry.

For more information about target function libraries in the build process, refer to “Introduction to Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation.

Using a Processor-Specific Target Function Library to Optimize Code

As a best practice, you should select the appropriate target function library for your processor after you verify the ANSI C implementation of your project.

Note Do not select the processor-specific target function library if you use your executable application on more than one specific processor. The operator and function entries in a library may work on more than one processor within a processor family. The entries in a library usually do not work with different processor families.

To use target function library for processor-specific optimization when you generate code, you must install Real-Time Workshop Embedded Coder software. Your model must include a Target Preferences block configured for you intended processor.

Perform the following steps to select the target function library for your processor:

- 1** Select **Simulation > Configuration Parameters** from the model menu bar. The Configuration Parameters dialog box for your model opens.
- 2** On the **Select** tree in the Configuration Parameters dialog box, choose **Real-Time Workshop**.
- 3** Use **Browse** to select as the **System target file**.
- 4** On the **Select** tree, choose **Interface**.

- 5** On the **Target function library** list, select the processor family that matches your processor. Then, click **OK** to save your changes and close the dialog box.

With the target function library selected, your generated code uses the specific functions in the library for your processor.

To stop using a processor-specific target function library, open the **Interface** pane in the model configuration parameters. Then, select the **C89/C90 (ANSI)** library from the **Target function library** list.

Process of Determining Optimization Effects Using Real-Time Profiling Capability

You can use the real-time profiling capability to examine the results of applying the processor-specific library functions and operators to your generated code. After you select a processor-specific target function library, use the real-time execution profiling capability to examine the change in program execution time.

Use the following process to evaluate the effects of applying a processor-specific target function library when you generate code:

- 1** Enable real-time profiling in your model. Refer to in the online Help system.
- 2** Generate code for your project using the default target function library **C89/C90 ANSI**.
- 3** Profile the code, and save the report.
- 4** Rebuild your project using a processor-specific target function library instead of the **C89/C90 ANSI** library.
- 5** Profile the code, and save the second report.
- 6** Compare the profile report from running your application with the processor-specific library selected to the profile results with the **ANSI** library selected in the first report.

Reviewing Processor-Specific Target Function Library Changes in Generated Code

Use one of the following techniques or tools to see the target function library elements where they appear in the generated code:

- Review the Code Manually.
- Use Model-to-Code Tracing to navigate from blocks in your model to the code generated from the block.
- Use a File Differencing Scheme to compare projects that you generate before and after you select a processor-specific target function library.

Reviewing Code Manually

To see where the generated code uses target function library replacements, review the file *modelName.c*. Look for code similar to the following statement

The function is the multiply implementation function registered in the target function library. In this example, the function performs an optimized multiplication operation. Similar functions appear for add, and sub. For more information about the arguments in the function, refer to “Introduction to Target Function Libraries” in the online Help system.

Using Model-to-Code Tracing

You can use the model-to-code report options in the configuration parameters to trace the code generated from any block with target function library. After you create your model and select a target function library, follow these steps to use the report options to trace the generated code:

- 1** Open the model configuration parameters.
- 2** Select **Report** from the **Select** tree.
- 3** In the **Report** pane, select **Create code generation report** and **Model-to-code**, and then save your changes.
- 4** Press **Ctrl+B** to generate code from your model.

The Real-Time Workshop Report window opens on your desktop. For more information about the report, refer to the Real-Time Workshop Embedded Coder documentation.

- 5 Use model-to-code highlighting to trace the code generated for each block with target function library applied:
 - Right-click on a block in your model and select **Real-Time Workshop > Navigate to code** from the context menu.
 - Select **Navigate-to-code** to highlight the code generated from the block in the report window.

Inspect the code to see the target function operator in the generated code. For more information, refer to “Tracing Code Generated Using Your Target Function Library” in the Real-Time Workshop Embedded Coder documentation in the online Help system.

If a target function library replacement did not occur as you expected, use the techniques described in “Examining and Validating Function Replacement Tables” in the Real-Time Workshop Embedded Coder documentation to help you determine why the build process did not use the function or operator.

Using a File Differencing Scheme

You can also review the target function library induced changes in your project by comparing projects that you generate both with and without the processor-specific target function library.

- 1 Generate your project with the default C89/C90 ANSI target function library. Use **Create Project**, **Archive Library**, or **Build** for the **Build action** in the Embedded IDE Link options.
- 2 Save the project to a new name—*newproject1*.
- 3 Go back to the configuration parameters for your model, and select a target function library appropriate for your processor.
- 4 Regenerate your project.
- 5 Save the project with a new name—*newproject2*

- 6 Compare the contents of the *modelName.c* files from `newproject1` and `newproject2`. The differences between the files show the target function library induced code changes.

Reviewing Target Function Library Operators and Functions

Real-Time Workshop Embedded Coder software provides the Target Function Library viewer to enable you to review the arithmetic operators and functions registered in target function library tables.

To open the viewer, enter the following command at the MATLAB prompt.

```
RTW.viewTf1
```

For details about using the target function library viewer, refer to “Selecting and Viewing Target Function Libraries” in the online Help system.

Creating Your Own Target Function Library

For details about creating your own library, refer to the following sections in your Real-Time Workshop Embedded Coder documentation:

- “Introduction to Target Function Libraries”
- “Creating Function Replacement Tables”
- “Examining and Validating Function Replacement Tables”

Model Reference

Model reference lets your model include other models as modular components. This technique provides useful features because it:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your Real-Time Workshop documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- Top model — The root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- Referenced models — Blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your Real-Time Workshop documentation in the online Help system.

Model Reference in Simulation

When you simulate the top model, Real-Time Workshop software detects that your model contains referenced models. Simulink software generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates

an executable (a MEX file, `.mex`) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink software rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these settings through the **Model Reference** pane of the Configuration Parameters dialog box.

Model Reference in Code Generation

Real-Time Workshop software requires executables to generate code from models. If you have not simulated your model at least once, Real-Time Workshop software creates a `.mex` file for simulation.

Next, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, Real-Time Workshop software calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

Using Model Reference

With few limitations or restrictions, Embedded IDE Link provides full support for generating code from models that use model reference.

Build Action Setting

The most important requirement for using model reference with the TI's processors is that you must set the **Build action** (go to **Configuration Parameters > Embedded IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action

- 1 Open your model.
- 2 Select **Simulation > Configuration Parameters** from the model menus.

The Configuration Parameters dialog box opens.

- 3** From the **Select** tree, choose **Embedded IDE Link**.
- 4** In the right pane, under **Runtime**, select set **Archive_library** from the **Build action** list.

If your top model uses a reference model that does not have the build action set to **Archive_library**, the build process automatically changes the build action to **Archive_library** and issues a warning about the change.

As a result of selecting the **Archive_library** setting, other options are disabled:

- DSP/BIOS is disabled for all referenced models. Only the top model supports DSP/BIOS operation.
- **Interrupt overrun notification method**, **Export IDE link handle to the base workspace**, and **System stack size** are disabled for the referenced models.

Target Preferences Blocks in Reference Models

Each referenced model and the top model must include a Target Preferences block for the correct processor. You must configure all the Target Preferences blocks for the same processor.

To obtain information about which compiler to use and which archiver to use to build the referenced models, the referenced models require Target Preferences blocks. Without them, the compile and archive processes does not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the Target Preferences block in the model provides.

Other Block Limitations

Model reference with Embedded IDE Link does not allow you to use certain blocks or S-functions in reference models:

- No blocks from the C62x DSP Library (in c6000lib) (because these are noninlined S-functions)
- No blocks from the C64x DSP Library (in c6000lib) (because these are noninlined S-functions)
- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Target Support Package™ or Target Support Package block library

Configuring processors to Use Model Reference

processors that you plan to use in Model Referencing must meet some general requirements.

- A model reference compatible processor must be derived from the ERT or GRT processors.
- When you generate code from a model that references another model, you need to configure both the top-level model and the referenced models for the same code generation processor.
- The External mode option is not supported in model reference Real-Time Workshop software processor builds. Embedded IDE Link does not support External mode. If you select this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities directory, as described in Supporting Shared Utility Directories in the Build Process in the Real-Time Workshop documentation.

To use an existing processor, or a new processor, with Model Reference, you set the `ModelReferenceCompliant` flag for the processor. For information on how to set this option, refer to `ModelReferenceCompliant` in the online Help system.

If you start with a model that was created prior to version 2.4 (R14SP3), to make your model compatible with the model reference processor, use the following command to set the `ModelReferenceCompliant` flag to On:

```
set_param(bdroot, 'ModelReferenceCompliant', 'on')
```

Models that you develop with versions 2.4 and later of Embedded IDE Link automatically include the model reference capability. You do not need to set the flag.

Verification

- “What Is Verification?” on page 4-2
- “Verifying Generated Code via Processor-in-the-Loop” on page 4-3
- “Profiling Code Execution in Real-Time” on page 4-9
- “System Stack Profiling” on page 4-17

What Is Verification?

Verification consists broadly of running generated code on a processor and verifying that the code does what you intend. The components of Embedded IDE Link combine to provide tools that help you verify your code during development by letting you run portions of simulations on your hardware and profiling the executing code.

Using the Automation Interface and Project Generator components, Embedded IDE Link offers the following verification functions:

- Processor-in-the-Loop — A technique to help you evaluate how your process runs on your processor
- Real-Time Task Execution Profiling — A tool that lets you see how the tasks in your process run in real-time on your processor hardware

Verifying Generated Code via Processor-in-the-Loop

In this section...

“What is Processor-in-the-Loop Cosimulation?” on page 4-3

“About the PIL Block” on page 4-4

“Preparing Your Model to Generate a PIL Application” on page 4-5

“Setting Model Configuration Parameters to Generate the PIL Application” on page 4-6

“Creating the PIL Block Application from a Model Subsystem” on page 4-6

“Running Your PIL Application to Perform Cosimulation and Verification” on page 4-7

“PIL Issues and Limitations” on page 4-7

What is Processor-in-the-Loop Cosimulation?

Processor in the loop (PIL) cosimulation is a technique to help you evaluate how well an algorithm, such as a control system or signal processing algorithm, operates on the processor selected for the application.

Note PIL requires Real-Time Workshop Embedded Coder software.

Cosimulation reflects a division of labor where Simulink software models the plant or test harness, while code generated from an algorithm in the model runs on the processor hardware.

During the Real-Time Workshop Embedded Coder software code generation process, you can create a PIL block from one of several Simulink software components including a model, a subsystem in a model, or subsystem in a library. You then place the generated PIL block inside a Simulink model that serves as the test harness and run tests to evaluate the processor-specific code execution behavior.

Definitions

PIL Algorithm

The algorithmic code, such as the signal processing algorithm, to test during the PIL cosimulation. The PIL algorithm is in compiled object form to enable verification at the object level.

PIL Application

The executable application that runs on the processor platform. The Embedded IDE Link creates a PIL application by augmenting your algorithmic code with the PIL execution framework. The PIL execution framework code is then compiled as part of your embedded application.

The PIL execution framework code includes the `string.h` header file so that the PIL application can use the `memcpy` function. The PIL application uses `memcpy` to exchange data between the Simulink model and the cosimulation processor.

PIL Block

A block you create from a subsystem in a model. When you run the simulation, the PIL block acts as the interface between the model and the PIL application running on the processor.

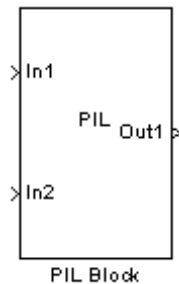
About the PIL Block

The PIL cosimulation block is the Simulink software block interface to PIL and the interface between the Simulink model and the executable PIL application running on the processor. Simulink model simulation inputs and outputs of the PIL cosimulation block match the input and output specification of the PIL algorithm.

The block is a basic building block that enables you to perform the following operations:

- Select a PIL algorithm
- Build and download a PIL application
- Run a PIL cosimulation

The PIL block inherits the shape and signal names from the source subsystem in your model, as shown in the following example. Inheritance is convenient for copying the PIL block into the model to replace the original subsystem for cosimulation.



Preparing Your Model to Generate a PIL Application

PIL verification begins with a model of the process to verify. Follow these steps to prepare your model to create a PIL application and PIL block:

1 Develop the model of the process to simulate.

Use Simulink software to build a model of the process to simulate. The blocks in the library can help you set up the timing and scheduling for your model.

For information about building Simulink software models, refer to *Getting Started with Simulink* in the online Help system.

2 Convert your process to a masked subsystem in your model.

For information about how to convert your process to a subsystem, refer to *Creating Subsystems* in *Using Simulink* or in the online Help system.

3 Open the new masked subsystem and add a Target Preferences block to the subsystem.

The block library contains the Target Preferences block to add to your system. Configure the Target Preferences block for your processor. For more information, refer to

Setting Model Configuration Parameters to Generate the PIL Application

After you create your subsystem, set the configuration parameters for your model to enable the model to generate a PIL block.

When you use PIL, you can set the configuration parameter **Solver options** to any selection from the **Type** and **Solver** lists.

Use the following steps:

- 1** Configure your model to enable it to generate PIL algorithm code and a PIL block from your subsystem.
 - a** From the model menu bar, go to **Simulation > Configuration Parameters** in your model to open the Configuration Parameters dialog box.
 - b** Choose **Real-Time Workshop** from the **Select** tree. Set the configuration parameters for your model as required by Embedded IDE Link software.
 - c** Under **Target selection**, set the **System target file** to .
- 2** Configure the model to perform PIL building and PIL block creation.
 - a** Select Embedded IDE Link on the **Select** tree.
 - b** On the **Build action** list, select `Create_processor_in_the_loop_project` to enable PIL.
 - c** Click **OK** to close the Configuration Parameters dialog box.

Creating the PIL Block Application from a Model Subsystem

Using PIL and PIL blocks to verify your processes begins with a Simulink model of your process. To see an example, refer to the demo Getting Started with Application Development in the demos for Embedded IDE Link.

Note Models can have multiple PIL blocks for different subsystems. They cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and incorrect results.

To create a PIL block, perform the following steps:

- 1 Right-click the masked subsystem in your model and select **Real-Time Workshop > Build Subsystem** from the context menu.

A new model window opens and the new PIL block appears in it.

This step builds the PIL algorithm object code and a PIL block that corresponds to the subsystem, with the same inputs and outputs. Follow the progress of the build process in the MATLAB command window.

- 2 Copy the new PIL block from the new model to your model, either in parallel to your masked subsystem to simulate the subsystem processes concurrently, or replace your subsystem with the PIL block.

To see the PIL block used in parallel to a masked subsystem, refer to the *Getting Started with Application Development* demo for your IDE among the demos.

Running Your PIL Application to Perform Cosimulation and Verification

After you add your PIL block to your model, click **Simulation > Start** to run the PIL simulation and view the results.

PIL Issues and Limitations

Consider the following issues when you work with PIL blocks.

Generic PIL Issues

Refer to the Support Table section in the Real-Time Workshop Embedded Coder documentation for general information about using the PIL block with embedded link products. Refer to PIL Feature Support and Limitations.

Real-Time Workshop grt.tlc-Based Targets Not Supported

Real-Time Workshop grt.tlc-based targets are not supported for PIL.

To use PIL, select the target file provided by Embedded IDE Link software.

Profiling Code Execution in Real-Time

In this section...

“Overview” on page 4-9

“Profiling Execution by Tasks” on page 4-10

“Profiling Execution by Subsystems” on page 4-12

Overview

Real-time execution profiling in Embedded IDE Link software uses a set of utilities to support profiling for synchronous and asynchronous tasks, or atomic subsystems, in your generated code. These utilities record, upload, and analyze the execution profile data.

Execution profiler supports profiling your code two ways:

- Tasks—Profile your project according to the tasks in the code.
- Atomic subsystems—Profile your project according to the atomic subsystems in your model.

Note To perform execution profiling, you must generate your project from a model in Simulink modeling environment.

When you enable profiling, you select whether to profile by task or subsystem.

To profile by subsystems, you must configure your model with at least one atomic subsystem. To learn more about creating atomic subsystems, refer to “Creating Subsystems” in the online help for Simulink software.

The profiler generates output in the following formats:

- Graphical display that shows task or subsystem activation, preemption, resumption, and completion. All data appears in a MATLAB graphic with the data notated by model rates or subsystems and execution time.

- An HTML report that provides statistical data about the execution of each task or atomic subsystem in the running process.

These reports are identical to the reports you see if you use `profile(ticcs_obj, 'execution', 'report')` to view the execution results. For more information about report formats, refer to `profile`. In combination, the reports provide a detailed analysis of how your code runs on the processor.

Use this general process for profiling your project:

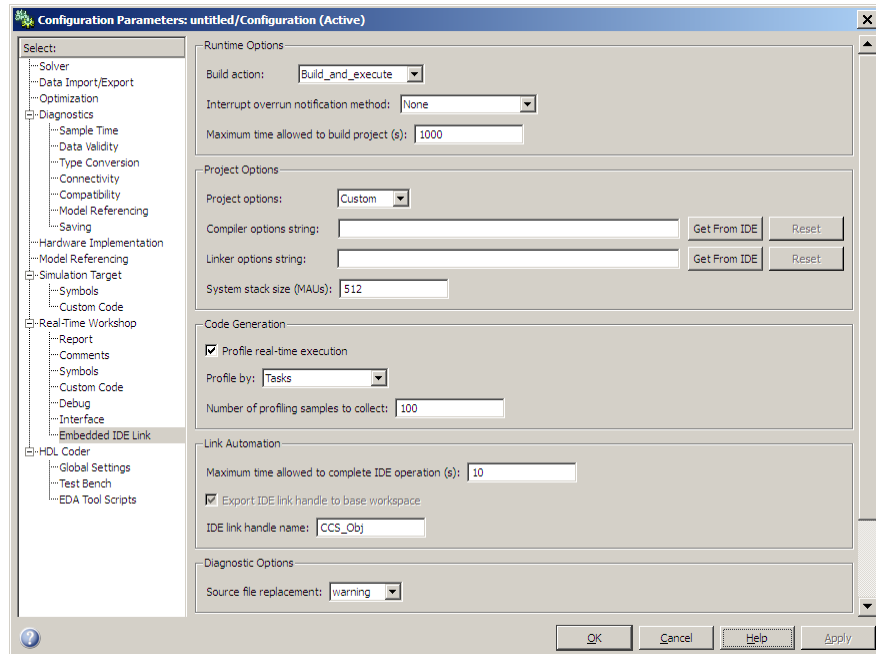
- 1** Create your model in Simulink modeling environment.
- 2** Enable execution profiling in the configuration parameters for your model.
- 3** Run your application.
- 4** Stop your application.
- 5** Get the profiling results with the `profile` function.

The following sections describe profiling your projects in more detail.

Profiling Execution by Tasks

To configure a model to use task execution profiling, perform the following steps:


- 1** Open the Configuration Parameters dialog box for your model.
- 2** Select Embedded IDE Link from the **Select** tree.
- 3** Select **Profile real-time execution**.
- 4** On the **Profile by** list, select **Tasks** to enable real-time task profiling.



5 By default, the **Export IDE link handle to base workspace** is enabled, and the **IDE link handle name** is set to `CCS_Obj`.

6 Click **OK** to close the Configuration Parameters dialog box.

To view the execution profile for your model:

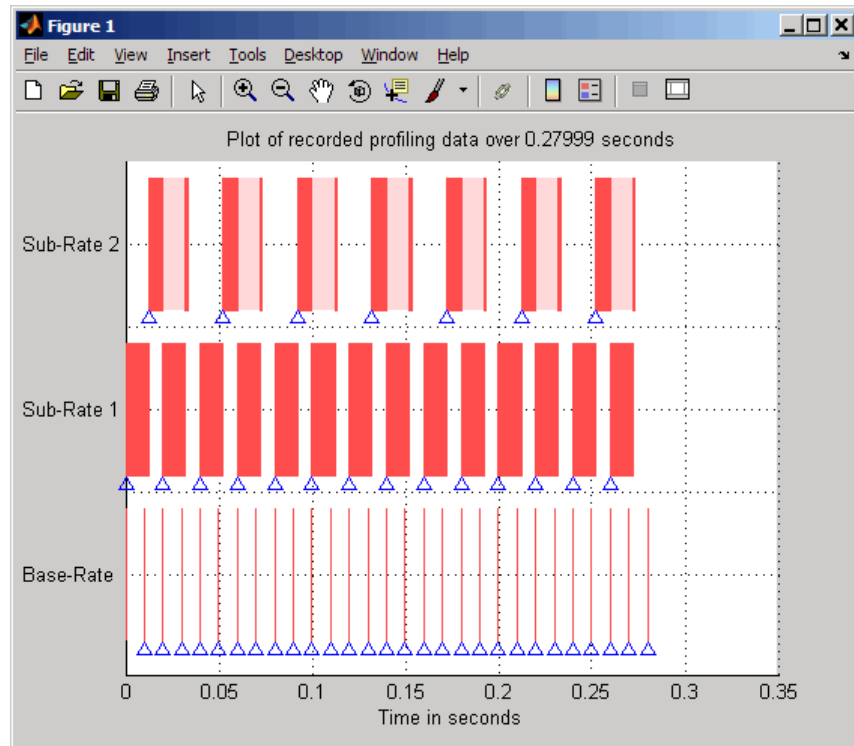
- 1** Click **Incremental build** () on the model toolbar to generate, build, load, and run your code on the processor.
- 2** To stop the running program, select **Debug > Halt** in CCS or use `halt(handlename)` from the MATLAB command prompt. Gathering profiling data from a running program may yield incorrect results.
- 3** At the MATLAB command prompt, enter

```
profile(handlename, execution , report )
```

to view the MATLAB software graphic of the execution report and the HTML execution report.

Refer to `profile` for information about other reporting options.

The following figure shows the profiling plot from running an application that has three rates—the base rate and two slower rates. The gaps in the Sub-Rate2 task bars indicate preempted operations.

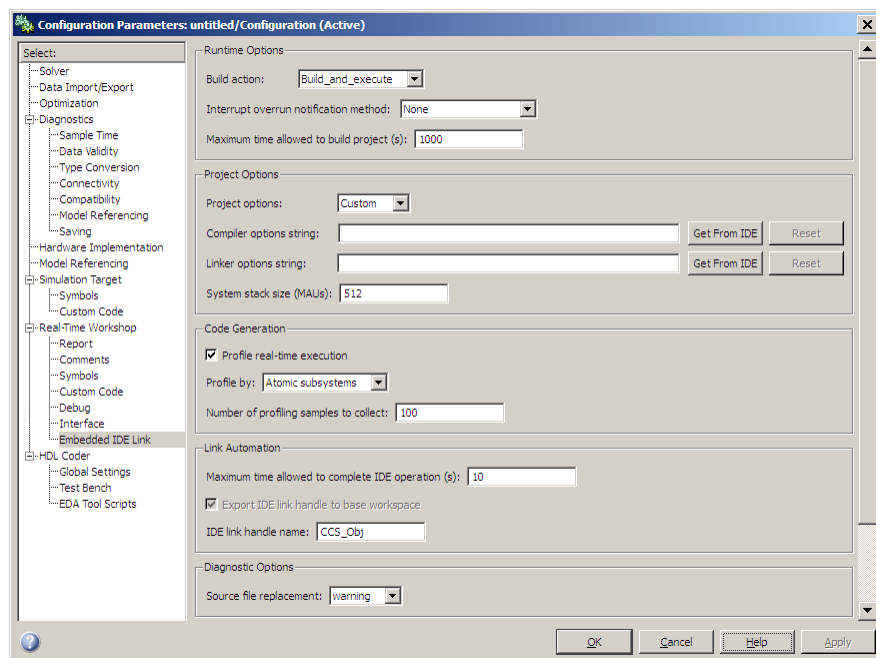


Profiling Execution by Subsystems

When your models use atomic subsystems, you have the option of profiling your code based on the subsystems along with the tasks.


To configure a model to use subsystem execution profiling, perform the following steps:

- 1 Open the Configuration Parameters dialog box for your model.
- 2 Select **Embedded IDE Link** from the **Select** tree. The pane appears as shown in the following figure.
- 3 Select **Profile real-time execution**.
- 4 On the **Profile by** list, select **Atomic subsystems** to enable real-time subsystem execution profiling.



- 5 By default, the **Export IDE link handle to base workspace** is enabled, and the **IDE link handle name** is set to `CCS_Obj`.
- 6 Click **OK** to close the Configuration Parameters dialog box.

To view the execution profile for your model:

- 1 Click **Incremental build** () on the model toolbar to generate, build, load, and run your code on the processor.

2 To stop the running program, select **Debug > Halt** in CCS, or use `halt(handlename)` from the MATLAB command prompt. Gathering profile data from a running program may yield incorrect results.

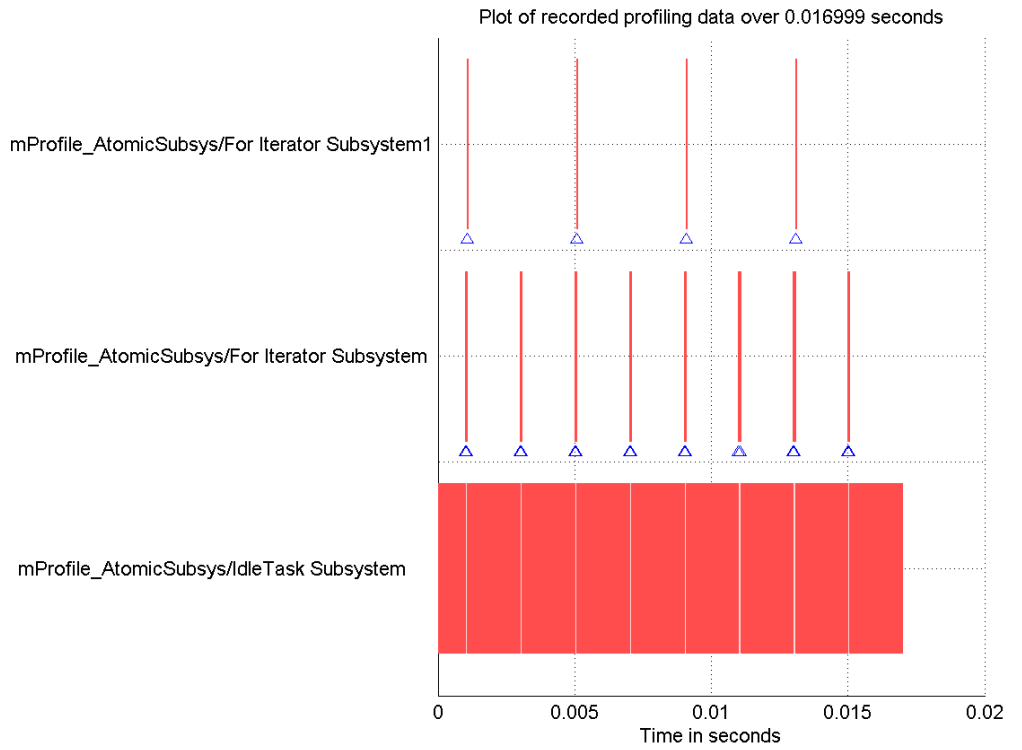
3 At the MATLAB command prompt, enter:

```
profile(handlename, execution , report )
```

to view the MATLAB software graphic of the execution report and the HTML execution report.

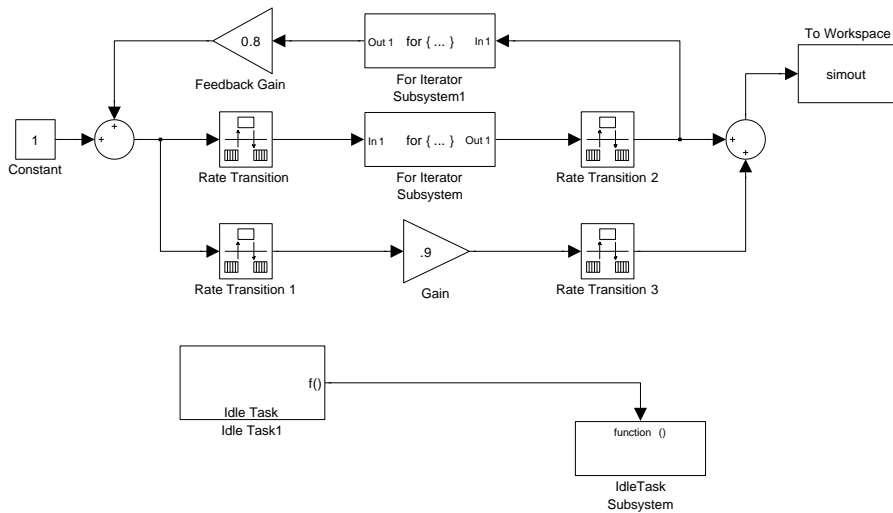
Refer to `profile` for more information.

The following figure shows the profiling plot from running an application that has three subsystems—For Iterator Subsystem, For Iterator Subsystem1, and Idle Task Subsystem.



The following figure presents the model that contains the subsystems reported in the profiling plot.

Atomic Subsystem Profiling



System Stack Profiling

In this section...
“Overview” on page 4-17
“Profiling System Stack Use” on page 4-19

Overview

Embedded IDE Link software enables you to determine how your application uses the processor system stack. Using the `profile` method, you can initialize and test the size and usage of the stack. This information can help you optimize both the size of the stack and how your code uses the stack.

To provide stack profiling, `profile` writes a known pattern to the addresses in the stack. After you run your application for a while, and then stop your application, `profile` examines the contents of the stack addresses. `profile` counts each address that no longer contains the known pattern as used. The total number of address that have been used, compared to the total number of addresses you allocated, becomes the stack usage profile. This profile process does not tell you how often any address was changed by your application.

You can profile the stack with both the hand written code in a project and the code you generate from a model.

Note Stack profiling always reports 100% stack usage when your project uses DSP/BIOS.

When you use `profile` to initialize and test the stack operation, the software returns a report that contains information about stack size, usage, addresses, and direction. With this information, you can modify your code to use the stack efficiently. The following program listing shows the stack usage results from running an application on a simulator.

```
profile(cc, 'stack', 'report')
```

```
Maximum stack usage:
```

System Stack: 532/1024 (51.95%) MAUs used.

```

        name: System Stack
    startAddress: [512    0]
      endAddress: [1535   0]
        stackSize: 1024 MAUs
    growthDirection: ascending
  
```

The following table describes the entries in the report:

Report Entry	Units	Description
System Stack	Minimum Addressable Unit (MAU)	Maximum number of MAUs used and the total MAUs allocated for the stack.
name	String for the stack name	Lists the name assigned to the stack.
startAddress	Decimal address and page	Lists the address of the stack start and the memory page.
endAddress	Decimal address and page	Lists the address of the end of the stack and the memory page.
stackSize	Addresses	Reports number of address locations, in MAUs, allocated for the stack.
growthDirection	Not applicable	Reports whether the stack grows from the lower address to the higher address (ascending) or from higher to lower (descending).

Profiling System Stack Use

To profile the system stack operation, perform these tasks in order:

- 1 Load an application.
- 2 Set up the stack to enable profiling.
- 3 Run your application.
- 4 Request the stack profile information.

Note If your application initializes the stack with known values when you run it, stack usage is reported as 100%. The value does not correctly reflect the stack usage. For example, DSP/BIOS™ writes a fixed pattern to the stack (0x00C0FFEE) when you run your project. This pattern prevents the stack profiler from reporting the stack usage correctly. Disable DSP/BIOS to use stack profiling in your project development.

Follow these steps to profile the stack as your application interacts with it. In this example, `cc` is an existing `ticcs` object.

- 1 Load the application to profile.
- 2 Use the `profile` method with the **setup** input keyword to initialize the stack to a known state.

```
profile(cc, 'stack', 'setup')
```

With the **setup** input argument, `profile` writes a known pattern into the addresses that compose the stack. For C6000 processors, the pattern is `A5`. For C2000 and C5000 processors, the pattern is `A5A5` to account for the address size. As long as your application does not write the same pattern to the system stack, `profile` can report the stack usage correctly.

- 3 Run your application.
- 4 Stop your running application. Stack use results gathered from an application that is running may be incorrect.

- 5** Use the `profile` method to capture and view the results of profiling the stack.

```
profile(cc, 'stack', 'report')
```

The following example demonstrates setting up and profiling the stack. The `ticcs` object `cc` must exist in your MATLAB workspace and your application must be loaded on your processor. This example comes from a C6713 simulator.

```
profile(cc, 'stack', 'setup') % Set up processor stack--write A5 to the stack addresses.
```

```
Maximum stack usage:
```

```
System Stack: 0/1024 (0%) MAUs used.
```

```
name: System Stack
startAddress: [512 0]
endAddress: [1535 0]
stackSize: 1024 MAUs
growthDirection: ascending
```

```
run(cc)
```

```
halt(cc)
```

```
profile(cc, 'stack', 'report') % Request stack use report.
```

```
Maximum stack usage:
```

```
System Stack: 356/1024 (34.77%) MAUs used.
```

```
name: System Stack
startAddress: [512 0]
endAddress: [1535 0]
stackSize: 1024 MAUs
growthDirection: ascending
```


Exporting Filter Coefficients from FDATool

- “About FDATool” on page 5-2
- “Preparing to Export Filter Coefficients to Code Composer Studio Projects” on page 5-4
- “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 5-9
- “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 5-15

About FDATool

Signal Processing Toolbox™ software provides the Filter Design and Analysis tool (FDATool) that lets you design a filter and then export the filter coefficients to a matching filter implemented in a CCS project.

Using FDATool with CCS IDE enables you to:

- Design your filter in FDATool
- Use CCS to test your filter on a processor
- Redesign and optimize the filter in FDATool
- Test your redesigned filter on the processor

For instructions on using FDATool, refer to the section “Filter Design and Analysis Tool” in the Signal Processing Toolbox documentation.

Procedures in this chapter demonstrate how to use the FDATool export options to export filter coefficients to CCS. Using these procedures, you can perform the following tasks:

- Export filter coefficients from FDATool in a header file—“Exporting Filter Coefficients from FDATool to the CCS IDE Editor” on page 5-9
- Export filter coefficients from FDATool to processor memory—“Replacing Existing Coefficients in Memory with Updated Coefficients” on page 5-16

Caution As a best practice, export coefficients in a header file for the most reliable results. Exporting coefficients directly to processor memory can generate unexpected results or corrupt memory.

Also see the reference pages for the following Embedded IDE Link functions. These primary functions allow you use to access variables and write them to processor memory from the MATLAB Command window.

- `address` — Return the address of a symbol so you can read or write to it.

- `ticcs` — Create a connection between MATLAB software and CCS IDE so you can work with the project in CCS from the MATLAB Command window.
- `write` — Write data to memory on the processor.

Preparing to Export Filter Coefficients to Code Composer Studio Projects

In this section...
“Features of a Filter” on page 5-4
“Selecting the Export Mode” on page 5-5
“Choosing the Export Data Type” on page 5-6

Features of a Filter

When you create a filter in FDATool, the filter includes defining features identified in the following table.

Defining Feature	Description
Structure	Structure defines how the elements of a digital filter—gains, adders/subtractors, and delays—combine to form the filter. See the Signal Processing Toolbox documentation in the Online Help system for more information about filter structures.
Design Method	Defines the mathematical algorithm used to determine the filter response, length, and coefficients.
Response Type and Specifications	Defines the filter passband shape, such as lowpass or bandpass, and the specifications for the passband.
Coefficients	Defines how the filter structure responds at each stage of the filter process.
Data Type	Defines how to represent the filter coefficients and the resulting filtered output. Whether your filter uses floating-point or fixed-point coefficients affects the filter response and output data values.

When you export your filter, FDATool exports only the number of and value of the filter coefficients and the data type used to define the coefficients.

Selecting the Export Mode

You can export a filter by generating an ANSI C header file, or by writing the filter coefficients directly to processor memory. The following table summarizes when and how to use the export modes.

To...	Use Export Mode...	When to Use	Suggested Use
Add filter coefficients to a project in CCS	C header file	You implemented a filter algorithm in your program, but you did not allocate memory on your processor for the filter coefficients.	<ul style="list-style-type: none"> • Add the generated ANSI C header file to an appropriate project. Building and loading this project into your processor allocates static memory locations on the processor and writes your filter coefficients to those locations. • Edit the file so the header file allocates extra processor memory and then add the header file to your project. Refer to “Allocating Sufficient or Extra Memory for Filter Coefficients” on page 5-15 in the next section. <p>(For a sample generated header file, refer to “Reviewing ANSI C Header File Contents” on page 5-12.)</p>
Modify the filter coefficients in an embedded application loaded on a processor	Write directly to memory	You loaded a program on your processor. The program allocated space in your processor memory to store the filter coefficients.	<ul style="list-style-type: none"> • Optimize your filter design in FDATool. <p>Then,</p> <ul style="list-style-type: none"> • Write the updated filter coefficients directly to the allocated processor memory. Refer to section “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 5-15 for more information.

Choosing the Export Data Type

The export process provides two ways you can specify the data type to use to represent the filter coefficients. Select one of the options shown in the following table when you export your filter.

Specify Data Type for Export	Description
Export suggested	Uses the data type that FDATool suggests to preserve the fidelity of the filter coefficients and the performance of your filter in the project
Export as	Lets you specify the data type to use to export the filter coefficients

FDATool exports filter coefficients that use the following data types directly without modifications:

- Signed integer (8, 16, or 32 bits)
- Unsigned integer (8, 16, or 32 bits)
- Double-precision floating point (64 bits)
- Single-precision floating point (32 bits)

Filters in FDATool in the Signal Processing Toolbox software use double-precision floating point. You cannot change the data type.

If you have installed Filter Design Toolbox™ software, you can use the filter quantization options in FDATool to set the word and fraction lengths that represent your filter coefficients. For information about using the quantization options, refer to Filter Design and Analysis Tool in the Filter Design Toolbox documentation in the Online help system.

If your filter uses one of the supported data types, **Export suggested** specifies that data type.

If your filter does not use one of the supported data types, FDATool converts the unsupported data type to one of the supported types and then suggests that data type. For more information about how FDATool determines the data

type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 5-7.

Follow these best-practice guidelines when you implement your filter algorithm in source code and design your filter in FDATool:

- Implement your filter using one of the data types FDATool exports without modifications.
- Design your filter in FDATool using the data type you used to implement your filter.

To Choose the Export Data Type

When you export your filter, follow this procedure to select the export data type to ensure the exported filter coefficients closely match the coefficients of your filter in FDATool.

- 1** In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog box.
- 2** Perform one of the following actions:
 - Select **Export suggested** to export the coefficients in the suggested data type.
 - Select **Export as** and choose the data type your filter requires from the list.

Caution If you select **Export as**, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

How FDATool Determines the Export Suggested Data Type

By default, FDATool represents filter coefficients as double-precision floating-point data. When you export your filter coefficients, FDATool suggests the same data type.

If you set custom word and fraction lengths to represent your filter coefficients, the export process suggests a data type to maintain the best fidelity for the filter.

The export process converts your custom word and fraction lengths to a suggested export data type, using the following rules:

- Round the word length up to the nearest larger supported data type. For example, round an 18-bit word length up to 32 bits.
- Set the fraction length to maintain the same difference between the word and fraction length in the new data type as applies in the custom data type.

For example, if you specify a fixed-point data type with word length of 14 bits and fraction length of 11 bits, the export process suggests an integer data type with word length of 16 bits and fraction length of 13 bits, retaining the 3 bit difference.

Exporting Filter Coefficients to Your Code Composer Studio Project

In this section...

“Exporting Filter Coefficients from FDATool to the CCS IDE Editor” on page 5-9

“Reviewing ANSI C Header File Contents” on page 5-12

Exporting Filter Coefficients from FDATool to the CCS IDE Editor

In this section, you export filter coefficients to a project by generating an ANSI C header file that contains the coefficients. The header file defines global arrays for the filter coefficients. When you compile and link the project to which you added the header file, the linker allocates the global arrays in static memory locations in processor memory.

Loading the executable file into your processor allocates enough memory to store the exported filter coefficients in processor memory and writes the coefficients to the allocated memory.

Use the following steps to export filter coefficients from FDATool to the CCS IDE text editor.

- 1 Start FDATool by entering `fdatool` at the MATLAB command prompt.

```
fdatool    % Starts FDATool.
```

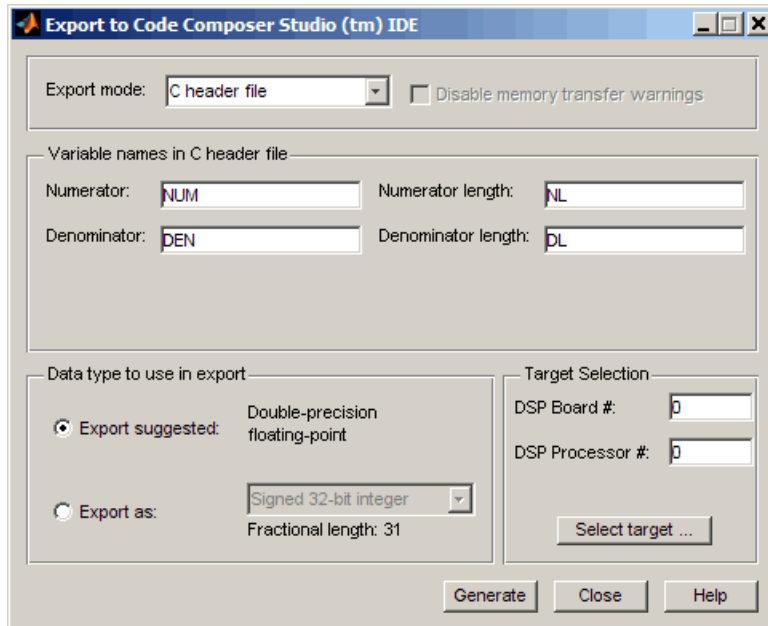
- 2 Design a filter with the same structure, length, design method, specifications, and data type you implemented in your source code filter algorithm.

The following figure shows a Direct-form II IIR filter example that uses second-order sections.

- 3 Click **Store Filter** to store your filter design. Storing the filter allows you to recall the design to modify it.

- 4 To export the filter coefficients, select **Targets > Code Composer Studio IDE** from the FDATool menu bar.

The Export to Code Composer Studio IDE dialog box opens, as shown in the following figure.



- 5 Set **Export mode** to **C header file**.



- 6 In **Variable names in C header file**, enter variable names for the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters where the coefficients will be stored.

The dialog box shows only the variables you need to export to define your filter.

Note You cannot use reserved ANSI C programming keywords, such as `if` or `int` as variable names, or include invalid characters such as spaces or semicolons (`;`).

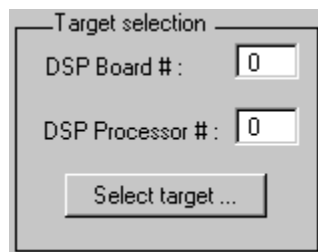
- 7** In **Data type to use in export**, select **Export suggested** to accept the recommended export data type. FDATool suggests a data type that retains filter coefficient fidelity.

You may find it useful to select the **Export as** option and select an export data type other than the one suggested.

Caution If you deviate from the suggested data type, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

For more information about how FDATool decides which data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 5-7.

- 8** If you know the board number and processor number of your DSP, enter **DSP Board #** and **DSP Processor #** values to identify your board.



The image shows a dialog box titled "Target selection". It contains two input fields: "DSP Board #:" with a text box containing the value "0", and "DSP Processor #:" with a text box containing the value "0". Below these fields is a button labeled "Select target ...".

When you have only one board or simulator, Embedded IDE Link software sets **DSP Board #** and **DSP Processor #** values for your board automatically.

If you have more than one board defined in CCS Setup:

- Click **Select target** to open the Selection Utility: Embedded IDE Link dialog box.
 - From the list of boards and list of processors, select the board name and processor name to use.
 - Click **Done** to set the **DSP Board #** and **DSP Processor #** values.
- 9 Click **Generate** to generate the ANSI header file. FDATool prompts you for a file name and location to save the generated header file.

The default location to save the file is your MATLAB working folder. The default file name is `fdacoefs.h`.

- 10 Click **OK** to export the header file to the CCS editor.

If CCS IDE is not open, this step starts the IDE.

The export process does not add the file to your active project in the IDE.

- 11 Drag your generated header file into the project that implements the filter.
- 12 Add a `#include` statement to your project source code to include the new header file when you build your project.
- 13 Generate a `.out` file and load the file into your processor. Loading the file allocates locations in static memory on the processor and writes the filter coefficients to those locations.

To see an example header file, refer to “Reviewing ANSI C Header File Contents” on page 5-12.

Reviewing ANSI C Header File Contents

The following program listing shows the exported header (`.h`) file that FDATool generates. This example shows a direct-form II filter that uses five second-order sections. The filter is stable and has linear phase.

Comments in the file describe the filter structure, number of sections, stability, and the phase of the filter. Source code shows the filter coefficients and variables associated with the filter design, such as the numerator length and the data type used to represent the coefficients.

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 *
 * Generated by MATLAB(R) 7.8 and the Signal Processing Toolbox 6.11.
 *
 * Generated on: xx-xxx-xxxx 14:24:45
 *
 */

/*
 * Discrete-Time IIR Filter (real)
 * -----
 * Filter Structure   : Direct-Form II, Second-Order Sections
 * Number of Sections : 5
 * Stable             : Yes
 * Linear Phase       : No
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"

/*
 * Expected path to tmwtypes.h
 * $MATLABROOT\extern\include\tmwtypes.h
 */
#define MWSPT_NSEC 11
const int NL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };
const real164_T NUM[MWSPT_NSEC][3] = {
    {
        0.802536131462,          0,          0
    },
    {
        0.2642710234701,    0.5285420469403,    0.2642710234701
    },
    {
        1,          0,          0
    },
    {
        0.1743690465012,    0.3487380930024,    0.1743690465012
    },
};
```

```
□  
  
    {  
        0.2436793028081,    0.4873586056161,    0.2436793028081  
    },  
    {  
        1,    0,    0  
    },  
    {  
        0.3768793219093,    0.7537586438185,    0.3768793219093  
    },  
    {  
        1,    0,    0  
    }  
};  
const int DL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };  
const rea164_T DEN[MWSPT_NSEC][3] = {  
    {  
        1,    0,    0  
    },  
    {  
        1,    -0.1842138030775,    0.1775781189277  
    },  
    {  
        1,    0,    0  
    },  
□ {  
        1,    -0.2160098642842,    0.3808329528195  
    },  
    {  
        1,    0,    0  
    }  
};
```

Preventing Memory Corruption When You Export Coefficients to Processor Memory

In this section...

“Allocating Sufficient or Extra Memory for Filter Coefficients” on page 5-15

“Example: Using the Exported Header File to Allocate Extra Processor Memory” on page 5-15

“Replacing Existing Coefficients in Memory with Updated Coefficients” on page 5-16

“Example: Changing Filter Coefficients Stored on Your Processor” on page 5-17

Allocating Sufficient or Extra Memory for Filter Coefficients

You can allocate extra memory by editing the generated ANSI C header file. You can then load the associated program file into your processor as described in “Example: Using the Exported Header File to Allocate Extra Processor Memory” on page 5-15. Extra memory lets you change filter coefficients and overwrite existing coefficients stored in processor memory more easily.

To prevent problems when you update filter coefficients in a project, , such as writing coefficients to unintended memory locations, use the `C header file export mode` option in FDATool to update filter coefficients in your program.

Example: Using the Exported Header File to Allocate Extra Processor Memory

You can edit the generated header file so the linked program file allocates extra processor memory. By allocating extra memory, you avoid the problem of insufficient memory when you export new coefficients directly to allocated memory.

For example, changing the following command in the header file:

```
const real64_T NUM[47] = {...}
```

to

```
real164_T NUM[256] = {...}
```

allocates enough memory for NUM to store up to 256 numerator filter coefficients rather than 47.

Exporting the header file to CCS IDE does not add the filter to your project. To incorporate the filter coefficients from the header file, add a `#include` statement:

```
#include "headerfilename.h"
```

Refer to “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 5-9 for information about generating a header file to export filter coefficients.

When you export filter coefficients directly to processor memory, the export process writes coefficients to as many memory locations as they need. The write process does not perform bounds checking. To ensure you write to the correct locations, and have enough memory for your filter coefficients, plan memory allocation carefully.

Replacing Existing Coefficients in Memory with Updated Coefficients

When you redesign a filter and export new coefficients to replace existing coefficients in memory, verify the following conditions for your new design:

- Your redesign did not increase the memory required to store the coefficients beyond the allocated memory.

Changes that increase the memory required to store the filter coefficients include the following redesigns:

- Increasing the filter order
 - Changing the number of sections in the filter
 - Changing the numerical precision (changing the export data type)
- Your changes did not change the export data type.

Caution Identify changes that require additional memory to store the coefficients before you begin your export. Otherwise, exporting the new filter coefficients may overwrite data in memory locations you did not allocate for storing coefficients. Also, exporting filter coefficients to memory after you change the filter order, structure, design algorithm, or data type can yield unexpected results and corrupt memory.

Changing the filter design algorithm in FDATool, such as changing from Butterworth to Maximally Flat, often changes the number of filter coefficients (the filter order), the number of sections, or both. Also, the coefficients from the new design algorithm may not perform properly with your source code filter implementation.

If you change the design algorithm, verify that your filter structure and length are the same after you redesign your filter, and that the coefficients will perform properly with the filter you implemented.

If you change the number of sections or the filter order, your filter will not perform properly unless your filter algorithm implementation accommodates the changes.

Example: Changing Filter Coefficients Stored on Your Processor

This example writes filter coefficients to processor memory to replace the existing coefficients. To perform this process, you need the names of the variables in which your project stores the filter data.

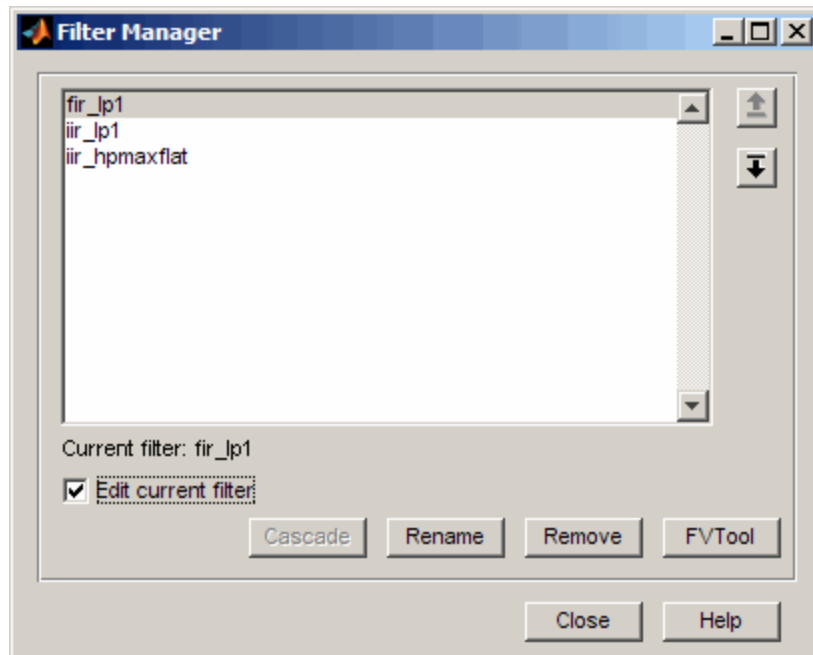
Before you export coefficients directly to memory, verify that your project allocated enough memory for the new filter coefficients. If your project allocated enough memory, you can modify your filter in FDATool and then follow the steps in this example to export the updated filter coefficients to the allocated memory.

If your new filter requires additional memory space, use a C header file to allocate memory on the processor and export the new coefficients as described in “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 5-9.

For important guidelines on writing directly to processor memory, refer to “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 5-15.

Follow these steps to export filter coefficients from FDATool directly to memory on your processor.

- 1 Load the program file that contains your filter into CCS IDE to activate the program symbol table. The symbol must contain the global variables you use to store the filter coefficients and length parameters.
- 2 Start FDATool.
- 3 Click **Filter Manager** to open the Filter Manager dialog box, shown in the following figure.



- 4 Highlight the filter to modify on the list of filters, and select **Edit current filter**. The highlighted filter appears in FDATool for you to change.

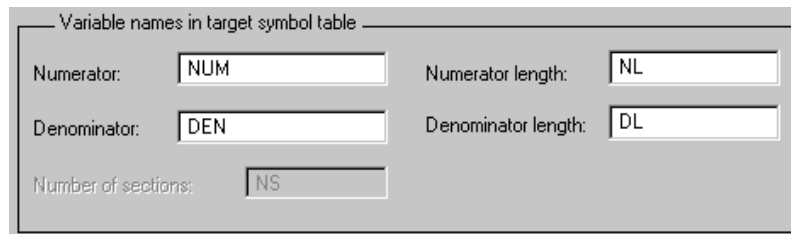
If you did not store your filter from a previous session, design the filter in FDATool and continue.

- 5 Click **Close** to dismiss the Filter Manager dialog box.
- 6 Adjust the filter specifications in FDATool to modify its performance.
- 7 In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog box.

Keep the export dialog box open while you work. When you do so, the contents update as you change the filter in FDATool.

Tip Click **Generate** to export coefficients to the same processor memory location multiple times without reentering variable names.

- 8 In the Export to Code Composer Studio dialog box:
 - Set **Export mode** to Write directly to memory
 - Clear **Disable memory transfer warnings** to get a warning if your processor does not support the export data type.
- 9 In **Variable names in target symbol table**, enter the names of the variables in the processor symbol table that correspond to the memory allocated for the parameters, such as **Numerator** and **Denominator**. Your names must match the names of the filter coefficient variables in your program.



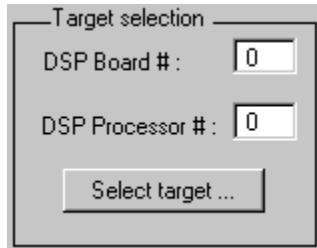
Variable names in target symbol table

Numerator:	<input type="text" value="NUM"/>	Numerator length:	<input type="text" value="NL"/>
Denominator:	<input type="text" value="DEN"/>	Denominator length:	<input type="text" value="DL"/>
Number of sections:	<input type="text" value="NS"/>		

- 10 Select **Export suggested** to accept the recommended export data type.

For more information about how FDATool determines the data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 5-7.

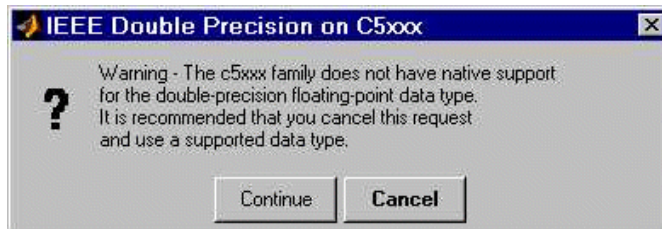
- 11 If you know the board number and processor number of your DSP, enter **DSP Board #** and **DSP Processor #** values to identify your board.



Note When you have only one board or simulator, Embedded IDE Link sets **DSP Board #** and **DSP Processor #** to your board automatically.

If you have more than one board defined in CCS Setup:

- Click **Select target** to open the Selection Utility: Embedded IDE Link dialog box.
 - Select the board name and processor name to use from the list of boards.
- 12 Click **Generate** to export your filter. If your processor does not support the data type you export, you see a warning similar to the following message.



You can continue to export the filter, or cancel the export process. To prevent this warning dialog box from appearing, select **Disable memory transfer warnings** in the Export to Code Composer Studio IDE dialog box.

- 13** (Optional) Continue to optimize filter performance by modifying your filter in FDATool and then export the updated filter coefficients directly to processor memory.
- 14** When you finish testing your filter, return to FDATool, and click **Store filter** to save your changes.

Function Reference

Operations on Objects for CCS IDE Work with links for CCS IDE
(p. 6-2)

Operations on Objects for RTDX Work with links to RTDX
(p. 6-4)

Operations on Objects for CCS IDE

<code>activate</code>	(For CCS) Activate specified CCS IDE project, file, or build configuration
<code>add</code>	(For CCS) Add files or new typedef to active project in CCS IDE
<code>address</code>	(For CCS) Return the memory address and page value for a symbol in CCS IDE
<code>animate</code>	(For CCS) Run application on processor to breakpoint
<code>build</code>	(For CCS) Build active project in CCS IDE
<code>ccsboardinfo</code>	(For CCS) Information about boards and simulators known to CCS IDE
<code>cd</code>	(For CCS) Change CCS IDE working folder
<code>close</code>	(For CCS) Close CCS IDE files or RTDX channel
<code>dir</code>	(For CCS) List files in current CCS IDE working directory
<code>display</code>	(For CCS) Display properties of object that refers to CCS IDE or RTDX link
<code>halt</code>	(For CCS) Terminate execution of process running on processor
<code>info</code>	(For CCS) Information about processor
<code>insert</code>	(For CCS) Add debug point to source file or address in CCS

<code>isreadable</code>	(For CCS) Determine whether MATLAB software can read specified memory block
<code>isrtdxcapable</code>	(For CCS) Determine whether processor supports RTDX
<code>isrunning</code>	(For CCS) Determine whether processor is executing process
<code>isvisible</code>	(For CCS) Determine whether CCS IDE is running
<code>iswritable</code>	(For CCS) Determine whether MATLAB software can write to specified memory block
<code>load</code>	(For CCS) Transfer program file (*.out, *.obj) to processor in active project
<code>new</code>	(For CCS) Create and open text file, project, or build configuration in CCS IDE
<code>open</code>	(For CCS) Open channel to processor or load file into CCS IDE
<code>profile</code>	(For CCS) Code execution and stack usage profile report
<code>read</code>	(For CCS) Data from memory on processor or in CCS
<code>regread</code>	(For CCS) Value from processor register
<code>regwrite</code>	(For CCS) Write data values to registers on processor
<code>reload</code>	(For CCS) Reload most recent program file to processor signal processor
<code>remove</code>	(For CCS) Remove file from active CCS IDE project

reset	(For CCS) Reset processor
restart	(For CCS) Restore program counter to entry point for current program
run	(For CCS) Execute program loaded on processor
symbol	(For CCS) Program symbol table from CCS IDE
ticcs	(For CCS) Create object that refers to CCS IDE
visible	(For CCS) Set whether CCS IDE window is visible while CCS runs
write	(For CCS) Write data to memory on processor

Operations on Objects for RTDX

close	(For CCS) Close CCS IDE files or RTDX channel
configure	(For CCS) Define size and number of RTDX channel buffers
disable	(For CCS) Disable RTDX interface, specified channel, or all RTDX channels
display	(For CCS) Display properties of object that refers to CCS IDE or RTDX link
enable	(For CCS) Enable RTDX interface, specified channel, or all RTDX channels
flush	(For CCS) Flush data or messages from specified RTDX channels

<code>isenabled</code>	(For CCS) Determine whether RTDX link is enabled for communications
<code>iswritable</code>	(For CCS) Determine whether MATLAB software can write to specified memory block
<code>msgcount</code>	(For CCS) Number of messages in read-enabled channel queue
<code>open</code>	(For CCS) Open channel to processor or load file into CCS IDE
<code>readmat</code>	(For CCS) Matrix of data from RTDX channel
<code>readmsg</code>	(For CCS) Read messages from specified RTDX channel
<code>writemsg</code>	(For CCS) Write messages to specified RTDX channel

Functions — Alphabetical List

activate

Purpose (For CCS) Activate specified CCS IDE project, file, or build configuration

Syntax `activate(cc, 'objectname', 'type')`

Description Use the `activate(cc, 'objectname', 'type')` method to make a project file, text file, or build configuration the active window in CCS IDE. After a project file, text file, and build configuration becomes active, you can apply other methods to it.

Inputs

`cc`

`cc` is a handle for an instance of CCS IDE. Before using the `activate` method, create `cc` using the `ticcs` function.

`objectname`

`objectname` is the project file, text file, or build configuration file the `activate` method makes active. For project and text files, enter the full file name including the extension. For build configurations, enter 'Debug', 'Release', or any custom configuration. Before using the `activate` method on a build configuration, you first activate the project that contains the build configuration.

`type`

`type` is the type of file `objectname` refers to. If you omit `type` from the `activate` method, `type` defaults to 'project'. Enter one of the following strings for `type`:

'project' — Indicates that `objectname` is a project file

'text' — Indicates that `objectname` is a text file

'buildcfg' — Indicates that `objectname` is a build configuration

Examples

This example demonstrates how to use `activate` to change the active project and document window.

After you create the projects, CCS IDE displays the active project in bold lettering in the project view. Similarly, after you create the new

build configuration, CCS IDE displays Testcfg as the active build configuration in myproject2.

```
cc=ticcs; %Create a handle for TI CCS.
visible(cc,1) %Make CCS IDE visible.
new(cc,'myproject1.pjt','project') %Create a new project.
new(cc,'myproject2.pjt') %If omitted, type defaults to 'project'.
    %myproject2 is active because you created it last.

new(cc,'Testcfg','buildcfg') %Create a build config in myproject2.

activate(cc,'myproject1.pjt','project') % Make myproject1 active.
add(cc,'c6711dsk_adc.c') %Add a .c file to myproject1.
activate(cc,'c6711dsk_adc.c','text') %Activate the .c file.
```

See Also

build, new, remove

add

Purpose (For CCS) Add files or new typedef to active project in CCS IDE

Syntax `add(cc, 'filename')`

Description `add(cc, 'filename')` adds a file to the active project in CCS IDE. When you add files, CCS automatically places them under the appropriate icon in the project view. For example, CCS IDE places `.c` source files under the Source icon and places `.lib` library files in the Libraries icon.

Using the add function equates selecting **Project > Add Files to Project** in CCS IDE.

Before using add:

- Create a handle, `cc`, for the CCS IDE using the `ticcs` command.
- Create a project, open a project, or make an existing project active in CCS IDE using the `new`, `open`, or `activate` methods.

You can use the add function to include in your project any of the file types shown in the following table.

File Types and Extensions Supported by add and CCS IDE

File Type	Extensions Supported	CCS Project Folder
C/C++ source files	<code>.c</code> , <code>.cpp</code> , <code>.cc</code> , <code>.ccx</code> , <code>.sa</code>	Source
Assembly source files	<code>.a*</code> , <code>.s*</code> (excluding <code>.sa</code> , refer to C/C++ source files)	Source
Object and library files	<code>.o*</code> , <code>.lib</code>	Libraries
Linker command file	<code>.cmd</code>	Project Name
DSP/BIOS file	<code>.tcf</code>	DSP/BIOS Config

Inputs

`add` places the file specified by *filename* in the active project in CCS.

cc

cc is a handle for an instance of CCS IDE. Before using the `add` method, create *cc* using the `ticcs` function.

filename

filename is the name of the file to add to the active CCS project.

If you supply a filename with no path or with a relative path, Embedded IDE Link searches the CCS IDE working folder first. It then searches the directories on your MATLAB path. Add supported file types shown in the preceding table.

Outputs

The `add` method assigns the `type`, `size`, and `uclass` of the file to `cc.type`.

Examples

Add files to a CCS IDE project.

```
cc=ticcs % Create a handle

TICCS Object:
  API version      : 1.2
  Processor type   : TMS320C64127
  Processor name   : CPU_1
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

cc.new('myproject','project'); % Create a new project.

cc.add('c6711dsk_adc.c'); % Add a C source file.
```

See Also

`activate`, `cd`, `new`, `open`, `remove`

address

Purpose (For CCS) Return the memory address and page value for a symbol in CCS IDE

Syntax `a = address(cc, 'symbolstring', 'varscope')`

Description The `a = address(cc, 'symbolstring', 'varscope')` method returns the memory address and page number of the first matching symbol in the symbol table of the most recently loaded program in CCS IDE.

The most recently loaded program in CCS IDE might not be the program loaded on the processor to which `cc` is linked.

Because the `address` method returns the `address` and `page` values as a structure, your programs can use the values directly. For example, the `cc.read` and `cc.write` can use `a` as an input.

If the `address` method does not find the symbol in the symbol table, it generates a warning and returns a null value.

Inputs

`a`

Use `a` as a variable to capture the return values from the `address` method.

`cc`

`cc` is a handle for an instance of CCS IDE. Before using the `address` method, create `cc` using the `ticcs` function.

`symbolstring`

`symbolstring` is the name of the symbol for which you are getting the memory address and page values.

Symbol names are case-sensitive. Use the proper case when you enter `symbolstring`

For `address` to work, `symbolstring` must represent a valid entry in the symbol table.

`varscope`

Optionally, you set the scope of the address method. Enter 'local' or 'global'.

Outputs

The `address` method returns the symbol name, memory address, and page values for the symbol as a 1-by-2 vector. The first cell contains the symbol name. The second cell contains the address and the memory page.

If the `address` method does not find the symbol, it returns the address as empty.

The `address` method only returns the first matching symbol in the symbol table.

The return value is a cell array where each row in a presents the symbol name and address in the table. This table shows a few possible elements of `a`, and their interpretation.

a Array Element	Contents of the Element
<code>a{1}</code>	String reflecting the symbol name. If address found a symbol that matches <i>symbolstring</i> , this is the same as <i>symbolstring</i> . Otherwise this is empty.
<code>a{2}(1)</code>	Address or value of symbol entry.
<code>a{2}(2)</code>	Memory page value. For TI's C6000 processors, the page is 0.

Examples

After you load a program to your processor, `address` lets you read and write to specific entries in the symbol table for the program. For example, the following function reads the value of symbol '`ddat`' from the symbol table in CCS IDE.

```
ddatv = read(cc,address(cc,'ddat'),'double',4)
```

`ddat` is an entry in the current symbol table. `address` searches for the string `ddat` and returns a value when it finds a match. `read` returns

address

ddat to MATLAB software as a double-precision value as specified by the string 'double'.

To change values in the symbol table, use `address` with `write`:

```
write(cc,address(cc,'ddat'),double([pi 12.3 exp(-1)...  
sin(pi/4)]))
```

After executing this write operation, *ddat* contains double-precision values for π , 12.3, e^{-1} , and $\sin(\pi/4)$. Use `read` to verify the contents of *ddat*:

```
ddatv = read(cc,address(cc,'ddat'),'double',4)
```

MATLAB software returns

```
ddatv =  
  
3.1416 12.3 0.3679 0.7071
```

See Also

`load`, `read`, `symbol`, `write`

Purpose (For CCS) Run application on processor to breakpoint

Syntax `animate(cc)`

Description `animate(cc)` starts the processor application, which runs until it encounters a breakpoint in the code. At the breakpoint, application execution halts and CCS Debugger returns data to CCS IDE to update all windows that are not connected to probe points. After updating the display, the application resumes execution and runs until it encounters another breakpoint. The run-break-resume process continues until you stop the application from MATLAB software with the `halt` function or from CCS IDE.

When you are running scripts or files in MATLAB software, you might find that `animate` provides a useful way to update the CCS IDE with information as your script or program runs.

Using `animate` with Multiprocessor Boards

When you use `animate` with a `ticcs` object `cc` that comprises more than one processor, such as an OMAP processor, the method applies to each processor in your `cc` object. This causes each processor to run a loaded program just as it does for the single processor case.

See Also `halt`, `restart`, `run`

build

Purpose (For CCS) Build active project in CCS IDE

Syntax

```
build(cc, timeout)
build(cc)
build(cc, 'all', timeout)
build(cc, 'all')
[result, numwarns]=build(...)
```

Description `build(cc, timeout)` incrementally rebuilds your active project in CCS IDE. In an incremental build:

- Files that you have changed since your last project build process get rebuilt or recompiled.
- Source files rebuild when the time stamp on the source file is later than the time stamp on the object file created by the last build.
- Files whose time stamps have not changed do not rebuild or recompile.

This incremental build is identical to the incremental build in CCS IDE, available from the CCS IDE toolbar.

After building the files, CCS IDE relinks the files to create the program file with the `.out` extension. To determine whether to relink the output file, CCS IDE compares the time stamp on the output file to the time stamp on each object file. It relinks the output when an object file time stamp is later than the output file time stamp.

To reduce the compile and build time, CCS IDE keeps a build information file for each project. CCS IDE uses this file to determine which file needs to be rebuilt or relinked during the incremental build. After each build, CCS IDE updates the build information file.

Note CCS IDE opens a Save As dialog box when the requested project build overwrites any files in the project. You must respond to the dialog box before CCS IDE continues the build. The dialog box may be hidden by open windows on your desktop and not visible. CCS IDE, MATLAB software, and other applications may appear to be frozen until you respond.

To limit the time that `build` spends performing the build, the optional argument `timeout` stops the process after `timeout` seconds. `timeout` defines the number of seconds allowed to complete the required compile, build, and link operation. If the build process exceeds the `timeout` period, `build` returns an error in MATLAB software. Generally, `build` causes the processor to initiate a restart even when the period specified by `timeout` passes. Exceeding the allotted time for the operation usually indicates that confirmation that the build was finished was not received before the `timeout` period passed. If you omit the `timeout` option in the syntax, `build` defaults to the global timeout defined in `cc`.

`build(cc)` is the same as `build(cc, timeout)` except that when you omit the `timeout` option, `build` defaults to the timeout for build, 1000 s. This timeout value overrides the default timeout setting for `cc`.

`build(cc, 'all', timeout)` completely rebuilds all of the files in the active project. This full build is identical to selecting **Project > Rebuild All** from the CCS menu bar. After rebuilding all files in the project, `build` performs the link operation to create a new program file.

To limit the time that `build` spends performing the build, optional argument `timeout` stops the process after `timeout` seconds. `timeout` defines the number of seconds allowed to complete the required compile, build, and link operation.

If the build process exceeds the timeout period, `build` returns an error in MATLAB software. Generally, `build` causes the processor to initiate a restart even when the period specified by `timeout` passes. Exceeding the allotted time for the operation usually indicates that confirmation that the build was finished was not received before the timeout period

build

passed. If you omit the *timeout* option in the syntax, `build` defaults to the global timeout defined in `cc`.

`build(cc, 'all')` is the same as `build(cc, 'all', timeout)` except that when you omit the *timeout* option, `build` defaults to the timeout set for build only, 1000 s.

`[result, numwarns]=build(...)` returns two output values that report the results of the build operation. For a successful build, the output arguments are the following:

- `result` equal to 1 for the build
- `numwarns` reports the number of build warnings that occurred during the build.

When the build is not successful, `build` displays an error and a message that contains the build string in the MATLAB software Command Window.

Examples

To demonstrate building a project from MATLAB software, use CCS IDE to load a project from the Texas Instruments software tutorials. For this example, open the project file `volume.pjt` from the `Tutorial` folder where you installed CCS IDE. (You can open any project you have for this example.)

Now use `build` to build the project:

```
cc=ticcs
```

```
TICCS Object:
```

```
API version      : 1.2
Processor type   : TMS320C64127
Processor name   : CPU_1
Running?        : No
Board number     : 0
Processor number : 0
Default timeout  : 10.00 secs
```



```
RTDX channels    : 0
```

```
build(cc, 'all', 20)
```

You just completed a full build of the project in CCS IDE. On the Build pane in CCS IDE, you see the record of the build process and the results. Now, make a change to a file in the project in CCS IDE and save the file. Then rebuild the project with an incremental build.

```
build(cc, 20)
```

When you look at the Build pane in CCS IDE, the log shows that the build only occurred on the file or files that you changed and saved.

See Also

activate, isrunning, open

ccsboardinfo

Purpose (For CCS) Information about boards and simulators known to CCS IDE

Syntax `ccsboardinfo`
`boards = ccsboardinfo`

Description `ccsboardinfo` returns configuration information about each board and processor installed and recognized by CCS. When you issue the function, `ccsboardinfo` returns the following information about each board or simulator.

Installed Board Configuration Data	Configuration Item Name	Description
Board number	boardnum	The number that CCS assigns to the board or simulator. Board numbering starts at 0 for the first board. This is also a property used when you create a new link to CCS IDE.
Board name	boardname	The name assigned to the board or simulator. Usually, the name is the board model name, such as TMS320C67xx evaluation module. If you are using a simulator, the name tells you which processor the simulator matches, such as C67xx simulator. If you renamed the board during setup, your assigned name appears here.

Installed Board Configuration Data	Configuration Item Name	Description
Processor number	procnum	The number assigned by CCS to the processor on the board or simulator. When the board contains more than one processor, CCS assigns a number to each processor, numbering from 0 for the first processor on the first board. For example, when you have two recognized boards, and the second has two processors, the first processor on the first board is procnum=0, and the first and second processors on the second board are procnum=1 and procnum=2. This is also a property used when you create a new link to CCS IDE.
Processor name	procname	Provides the name of the processor. Usually the name is CPU, unless you assign a different name.
Processor type	proctype	Gives the processor model, such as TMS320C6x1x for the C6xxx series processors.

Each row in the table that you see displayed represents one digital signal processor, either on a board or simulator. As a consequence, you use the information in the table in the function `ticcs` to identify a selected board in your PC.

`boards = ccsboardinfo` returns the configuration information about your installed boards in a slightly different manner. Rather than returning the table containing the information, you get a listing of the board names and numbers, where each board has an associated structure named `proc` that contains the information about each processor on the board. For example

```
boards = ccsboardinfo
```

returns

```
boards =  
  
    name: 'C6xxx Simulator (Texas Instruments)'  
    number: 0  
    proc: [1x1 struct]
```

where the structure `proc` contains the processor information for the C6xxx simulator board:

```
boards.proc  
  
ans =  
  
    name: 'CPU'  
    number: 0  
    type: 'TMS320C6200'
```

Reviewing the output from both function syntaxes shows that the configuration information is the same.

When you combine this syntax with the dot notation used to access the elements in a structure, the result is a way to determine which board to connect to when you construct a link to CCS IDE. For example, when you are creating a link to a board in your PC, the dot notation provides the means to set the board by issuing the command with the `boardnum` and `procnum` properties set to the entries in the structure `boards`. For example, when you enter

```
boards = ccsboardinfo;
```

`boards(1).name` returns the name of your second installed board and `boards(1).proc(2).name` returns the name of the second processor on the second board. To create a link to the second processor on the second board, use

```
cc = ticcs('boardnum',boards(1).number,'procnum',...
```

```
boards(1).proc(2).name);
```

Examples

On a PC with both a simulator and a DSP Starter Kit (DSK) board installed,

```
ccsboardinfo
```

returns something similar to the following table. Your display may differ slightly based on what you called your boards when you configured them in CCS Setup Utility:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum ..	0	CPU	TMS320C6200
0	DSK (Texas Instruments)	0	CPU_3	TMS320C6x1x

When you have one or more boards that have multiple CPUs, ccsboardinfo returns the following table, or one similar to it:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
2	C6xxx Simulator (Texas Instrum .0	0	CPU	TMS320C6200
1	C6xxx EVM (Texas Instrum ...	1	CPU_Primary	TMS320C6200
1	C6xxx EVM (Texas Instrum ...	0	CPU_Secondary	TMS320C6200
0	C64xx Simulator (Texas Instru...0	0	CPU	TMS320C64xx

In this example, board number 1 returns two defined CPUs: CPU_Primary and CPU_Secondary. The C6xxx does not in fact have two CPUs; a second CPU is defined for this example.

To demonstrate the syntax `boards = ccsboardinfo`, this example assumes a PC with two boards installed, one of which has three CPUs.

Enter

```
ccsboardinfo
```

at the MATLAB desktop prompt. You get

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum	.0	CPU	TMS320C6211
0	C6211 DSK (Texas Instruments)	2	CPU_3	TMS320C6x1x
0	C6211 DSK (Texas Instruments)	1	CPU_4_1	TMS320C6x1x
0	C6211 DSK (Texas Instruments)	0	CPU_4_2	TMS320C6x1x

Now enter

```
boards = ccsboardinfo
```

MATLAB software returns

```
boards=  
2x1 struct array with fields  
    name  
    number  
    proc
```

showing that you have two boards in your PC.

Use the dot notation to determine the names of the boards:

```
boards.name
```

returns

```
ans=  
C6xxx Simulator (Texas Instruments)
```

```
ans=  
C6211 DSK (Texas Instruments)
```

To identify the processors on each board, again use the dot notation to access the processor information. You have two boards (numbered 0 and

1). Board 0 has three CPUs defined for it. To determine the type of the second processor on board 0 (the board whose boardnum = 0), enter

```
boards(2).proc(1)
```

which returns

```
ans=
  name: 'CPU_3'
  number: 1
  type: 'TMS320C6x1x'
```

Recall that

```
boards(2).proc
```

gives you this information about the board

```
ans=
3x1 struct array with fields:
  name
  number
  type
```

indicating that this board has three processors (the 3x1 array).

The dot notation is useful for accessing the contents of a structure when you create a link to CCS IDE. When you use `ticcs` to create your CCS link, you can use the dot notation to tell CCS IDE which processor you are using.

```
cc = ticcs('boardnum',boards(1).proc(1))
```

See Also

`info`, `ticcs`

Purpose (For CCS) Change CCS IDE working folder

Syntax

```
cd(cc, 'directory')  
wd = cd(cc, 'directory')  
cd(cc, pwd)
```

Description `cd(cc, 'directory')` changes the CCS IDE working directory to the directory identified by the string *directory*. For the change to take effect, *directory* must refer to an existing directory. You can give the directory string either as a relative path name or an absolute path name including the drive letter. CCS IDE applies relative path names from the current working directory.

`wd = cd(cc, 'directory')` returns the current CCS IDE working directory in `wd`.

Using `cc` to change the CCS IDE working directory does not affect your MATLAB environment working directory or any MATLAB environment paths. Use the following function syntax to set your CCS IDE working directory to match your MATLAB environment working directory.

`cd(cc, pwd)` where `pwd` calls the MATLAB function `pwd` that shows your present MATLAB working directory and changes your current CCS IDE working directory to match the path name returned by `pwd`.

Examples When you open a project in CCS IDE, the folder containing the project becomes the current working folder in CCS IDE. Try opening the tutorial project `volume.mak` in CCS IDE. `volume.mak` is in the tutorial files from CCS IDE. When you check the working directory for CCS IDE in the MATLAB environment, you see something like the following result

```
wd=cd(cc)  
  
wd =  
  
D:\ticcs\c6000\tutorial\volume1
```


where the drive letter D may be different based on where you installed CCS IDE.

Now check your MATLAB environment working directory:

```
pwd
```

```
ans =
```

```
J:\bin\win32
```

Your CCS IDE and MATLAB environment working directories are not the same. To make the directories the same, use the `cd(cc,pwd)` syntax:

```
cd(cc,pwd) % Set CCS IDE to use your MATLAB working directory.
```

```
pwd % Check your MATLAB working directory.
```

```
ans =
```

```
J:\bin\win32
```

```
cd(cc) % Check your CCS IDE working directory.
```

```
ans =
```

```
J:\bin\win32
```

You have set CCS IDE and MATLAB environment to use the same working directory.

See Also

`dir`, `load`, `open`

close

Purpose (For CCS) Close CCS IDE files or RTDX channel

Note `close(cc, 'filename', 'text')` produces an error. Support for `close(rx, ...)` on C5000 and C6000 processors will be removed in a future version.

Syntax

```
close(cc, 'filename', 'type')
close(rx, 'channel1', 'channel2', ...)
close(rx, 'channel')
```

Description `close(cc, 'filename', 'type')` closes the file in CCS IDE identified by *filename* of type *type*. *type* identifies the type of file to close. This can be either project files when you use 'project' for the *type* option, or text files when you use 'text' for the *type* option. To close a specific file in CCS IDE, *filename* must match exactly the name of the file to close. If you replace *filename* with 'all', `close` terminates every open file whose type matches the *type* option. File types recognized by `close` include these extensions.

type String	Affected files
'project'	Project files with the .pj1 extension.
'text'	All files with these extensions — .a*, .c, .cc, .ccx, .tcf, .cmd, .cpp, .lib, .o*, .rcp, and .s*. Note that 'text' does not close .cfg files.

When you replace *filename* with the null entry [], `close` shuts the current active file window in CCS IDE. When you specify 'project' for the *type* option, it closes the active project.

Note `close` does not save files before shutting them. Closing files can result in lost data if you have changed the files after you last saved them. Use `save` to preserve your changes before you close files.

`close(rx, 'channel1', 'channel2', ...)` closes the channels specified by the strings `channel1`, `channel2`, and so on as defined in `rx`.

`close(rx, 'channel')` closes the specified channel. When you set `channel` to `'all'`, this function closes all the open channels associated with `rx`.

To avoid conflicts, do not name channels “all” or “ALL.”

Examples

Using close with Files and Projects

To clarify the different `close` options, here are six commands that close open files or projects in CCS IDE.

Command	Result
<code>close(cc, 'all', 'project')</code>	Close all open projects in CCS IDE.
<code>close(cc, 'my.pjt', 'project')</code>	Close the project <code>my.pjt</code> .
<code>close(cc, [], 'project')</code>	Close the active project.
<code>close(cc, 'all', 'text')</code>	Close all open text files. This includes source file, libraries, command files, and others.
<code>close(cc, 'my_source.cpp', 'text')</code>	Close the text file <code>my_source.cpp</code> .
<code>close(cc, [], 'text')</code>	Close the active file window.

Using close with RTDX

When you plan to use RTDX to communicate with a processor, you open and enable channels to the board and processor. For example, to communicate with the processor on your installed board, you use `open` to set up a channel, as follows:

```
cc = ticcs('boardnum',1,'procnum',0)
```

close

```
rx=cc.rtdx % Create an alias to the RTDX portion of this link.  
open(rx,'ichan','w') % Open a channel for write access.  
enable(rx,'ichan') % Enable the open channel for use.
```

After you finish using the open channel, you must close it to avoid difficulties later on.

```
close(rx,'ichan')
```

Or to close all open channels, you could use

```
close(rx,'all')
```

See Also

disable, open

Purpose (For CCS) Define size and number of RTDX channel buffers

Note `configure` produces a warning on C5000 and C6000 processors and will be removed in a future version.

Syntax `configure(rx,length,num)`

Description `configure(rx,length,num)` sets the size of each main (host) buffer, and the number of buffers associated with `rx`. Input argument `length` is the size in bytes of each channel buffer and `num` is the number of channel buffers to create.

Main buffers must be at least 1024 bytes, with the maximum defined by the largest message. On 16-bit processors, the main buffer must be four bytes larger than the largest message. On 32-bit processors, set the buffer to be eight bytes larger than the largest message. By default, `configure` creates four, 1024-byte buffers. Independent of the value of `num`, CCS IDE allocates one buffer for each processor.

Use CCS to check the number of buffers and the length of each one.

Examples Create a default link to CCS and configure six main buffers of 4096 bytes each for the link.

```
cc=ticcs           % Create the CCS link with default values.
```

```
TICCS Object:
```

```
API version       : 1.0
Processor type    : C67
Processor name    : CPU
Running?         : No
Board number      : 0
Processor number  : 0
Default timeout   : 10.00 secs

RTDX channels     : 0
```

configure

```
rx=cc.rtdx           % Create an alias to the rtdx portion.  
  
RTDX channels      : 0  
  
configure(rx,4096,6) % Use the alias rx to configure the length  
                    % and number of buffers.
```

After you configure the buffers, use the RTDX tools in CCS IDE to verify the buffers.

See Also

readmat, readmsg, write, writemsg

Purpose

(For CCS) Open Data Type Manager

`datatypemanager` produces a warning and will be removed in a future version.

Syntax

```
datatypemanager(cc)  
cc2 = datatypemanager(cc)
```

Description

`datatypemanager(cc)` opens the Data Type Manager (DTM) with data type information about the project to which `cc` refers. With the type manager open, you can add type definitions (typedefs) from your project to MATLAB software so it can interpret them. You add your typedefs because MATLAB software cannot determine or understand typedefs in your function prototypes remotely across the interface to CCS.

Each custom type definition in your prototype must appear on the **Typedef name (Equivalent data type)** list before you can use the typedef from MATLAB software with a function object.

When the DTM opens, a variety of information and options displays in the Data Type Manager dialog box:

- **Typedef name (Equivalent data type)** — provides a list of default data types. When you create a typedef, it appears added to this list.
- **Add typedef** — opens the **Add Typedef** dialog box so you can add one or more typedefs to your project. Your added typedef appears on the **Typedef name (Equivalent data type)** list. Also, when you pass the `cc` object to the DTM, and then add a typedef, the command

```
cc.type
```

returns a list of the data types in the object including the typedefs you added.

- **Remove typedef** — removes a selected typedef from the **Typedef name (Equivalent data type)** list.
- **Load session** — loads a previously saved session so you can use the typedefs you defined earlier without reentering them.

- **Refresh list** — updates the list in **Typedef name (Equivalent data type)**. Refreshing the list ensures the contents are current. If you changed your project data type content or loaded a new project, this updates the type definitions in the DTM.
- **Close** — closes the DTM and prompts you to save the session information. This is the only way to save your work in this dialog box. Saving the session creates an M-file you can reload into the DTM later.

Clicking **Close** in the DTM prompts you to save your session. Saving the session creates an M-file that contains operations that create your final list of data types, identical to the data types in the **Typedef name** list.

In the stored M-file, you find a function that includes the add and remove operations you used to create the list of data types in the DTM. For each time you added a typedef in the DTM, the M-file contains an add command that adds the new type definition to the `cc.type` property of the object. When you remove a data type, you see an equivalent `clear` command that removes a data type from the `cc.type` object.

Note All operations that add and remove data types in the DTM during a session are stored in the generated M-file, including mistakes you make while creating or removing type definitions. When you load your saved session into the DTM, you see the same error messages you saw during the session. Keep in mind that you have already corrected these errors.

The first line of the M-file is a function definition, where the name of the function is the filename of the session you saved.

`cc2 = datatypemanager(cc)` returns the `cc2` ticcs object while it opens the DTM. `cc2` represents an alias to `cc`. Objects `cc` and `cc2` are not

independent objects. When you change a property of either `cc` or `cc2`, the corresponding property in the other object changes as well.

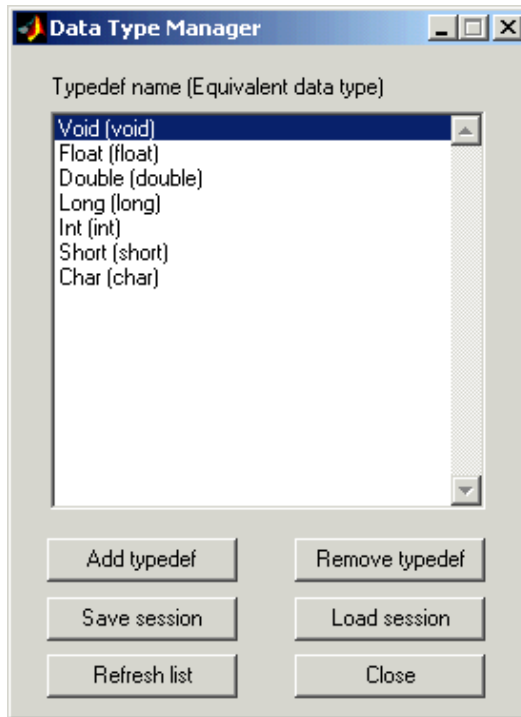
Data Type Manager

When you create objects that access functions in a project, MATLAB software can recognize most data types that you use in your project. However, if the functions use one or more custom type definitions, MATLAB software cannot recognize the data type and cannot work with the function. To overcome this problem, the Data Type Manager provides the capability to define your typedefs to MATLAB software.

Entering

```
datatypemanager(cc)
```

at the MATLAB prompt opens the DTM.



Before you add a type definition, the **Typedef name (Equivalent data type)** list shows a number of data types already defined:

- `Void(void)` — void return argument for a function
- `Float(float)` — float data type used in a function input or return argument
- `Double(double)` — double data type used in a function input or return argument
- `Long(long)` — long data type used in a function input or return argument
- `Int(int)` — int data type used in a function input or return argument

- `Short` (`short`) — short data type used in a function input or return argument
- `Char` (`char`) — character data type used in a function input or return argument

The lowercase versions of the data types appear because MATLAB software does not recognize the initial capital versions automatically. In the data type entry, the project data type with the initial capital letter is mapped to the lowercase MATLAB software data type.

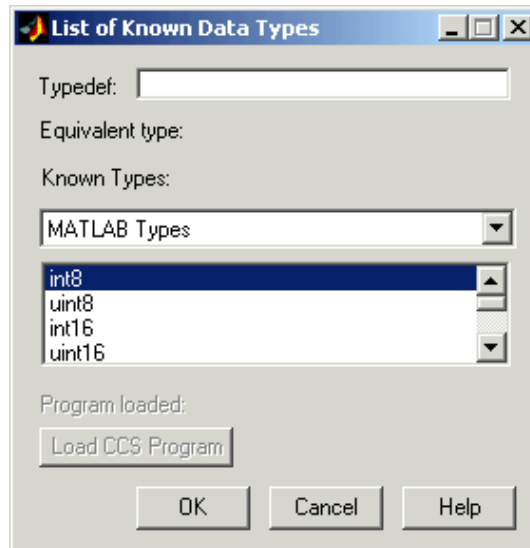
Although not recommended, you can use mixed case typedef names, so long as the equivalent data type uses lowercase. In particular, typedefs that refer to other typedefs should resolve to a data type in lowercase.

Adding a type definition adds the new data type to the list of typedefs.

Remove any existing or new type definitions with the **Remove typedef** option.

Add Typedef Dialog Box

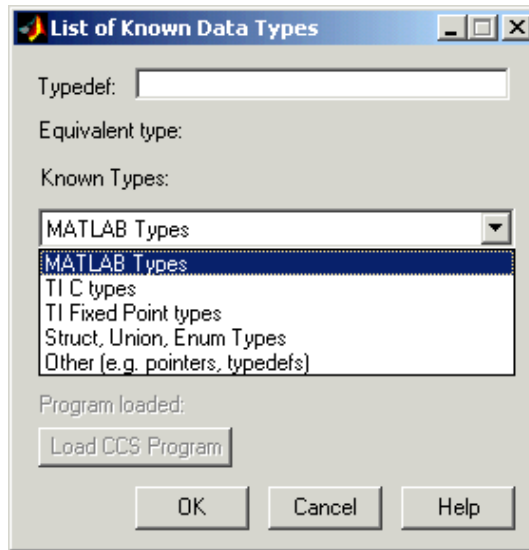
Clicking **Add typedef** in the DTM opens the List of Known Data Types dialog box. As shown in this figure, you add your custom type definitions here.



When you have used custom type definitions in your program or project, you must specify what they mean to MATLAB software. The **Typedef** option lets you enter the name of the typedef in your program and select an equivalent type from the **Known Types** list. By defining your type definitions in this dialog box, you enable MATLAB software to understand and work with them. For example, when you return the data to the MATLAB workspace or send data from the workspace to your project.

After you define each typedef, the **Equivalent type** option shows you the type you specified for each type definition, either when you enter it in the **Typedef** field or select it from the **Known Types** list.

Options in this dialog box let you review the data types you are using or that are available in your projects. By selecting different data type categories from the **Known Types** list, you can see all of the supported data types.



From the list of known data types, choose one of the following data type categories:

- MATLAB Types

Data Type	Description
int8	8-bit integer data
uint8	Unsigned 8-bit integer data
int16	16-bit integer data
uint16	Unsigned 16-bit integer data
int32	32-bit integer data

datatypemanager

Data Type	Description
uint32	Unsigned 32-bit integer data
int64	64-bit integer data
uint64	Unsigned 64-bit integer data
single	32-bit IEEE® floating-point data
double	64-bit IEEE floating-point data

- TI C Types

Data Type	Description (For C6000 Compiler)
char	8-bit character data with a sign bit
unsigned char	8-bit character data
signed char	8-bit character data
short	16-bit numeric data
unsigned short	Unsigned 16-bit numeric data
signed short	16-bit numeric data with sign designation
int	32-bit integer numeric data
unsigned int	32-bit integer numerics without sign information
signed int	32-bit integer numerics with sign information
long	40-bit data with sign bit. Note that this is not the same as int.
unsigned long	40-bit data without information about the sign of the number
signed long	40-bit data without information about the sign of the number represented
float	32-bit numeric data

Data Type	Description (For C6000 Compiler)
double	64-bit numeric data
long double	On the C2000 and C5000 processors – 32-bit IEEE floating-point data On the C6000 processors – 64-bit IEEE floating-point data

Numbers of bits change depending on the processor and compiler. For more information about Texas Instruments data types and specific processors or compilers, refer to your compiler documentation from Texas Instruments processors.

- TI Fixed-Point Types

Data Type	Description
Q0.15	Numeric data with 16-bit word length and 15-bit fraction length
Q0.31	32-bit word length numeric data with fraction length of 31 bits

- Struct, Union, Enum types

If the program you load on the processor includes one or more of `struct`, `union`, or `enum` data types, the type definitions show up on this list. Until you load a program on the processor, this list is empty and trying to access the list generates an error message.

Load a program, if you have not already done so, by clicking **Load CCS Program** and selecting a `.out` file to load on your processor.

- When the load process works, you see the name of the file you loaded in **Loaded program**. Otherwise you get an error message that the load failed.

Only programs that you load from this dialog box appear in **Program loaded**. Programs that you already loaded on your processor do not

appear in the **Loaded program** option. MATLAB software cannot determine what program you have loaded.

- Others such as pointers and typedefs

Like `struct`, `union`, and `enum` data types, the **Others** list is empty until you define one or more typedefs. Unlike the **Struct**, **Union**, **Enum types** list, loading a program does not populate this list with typedefs from the program. You must define them explicitly in this dialog box.

Custom type definitions can refer to other typedefs in your project. Nesting typedefs works after you have define the necessary custom types. To create a typedef that uses another typedef, define the nested (inner) definition, and then define the outer definition as a pointer to the nested definition. Refer to **Examples** to see this in operation.

Program loaded — tells you the name of the program loaded on the processor, if you loaded the program from this dialog box. If not, **Program loaded** does not report the program name.

Load CCS Program — opens the **Load Program** dialog box so you can select and load a `.out` file to your processor.

Examples

This set of examples show how to create custom type definitions with the DTM. Each example shows the **List of Known Data Types** dialog box with the selections or entries needed to create the typedef.

Start the examples by creating a `ticcs` object:

```
cc=ticcs;
```

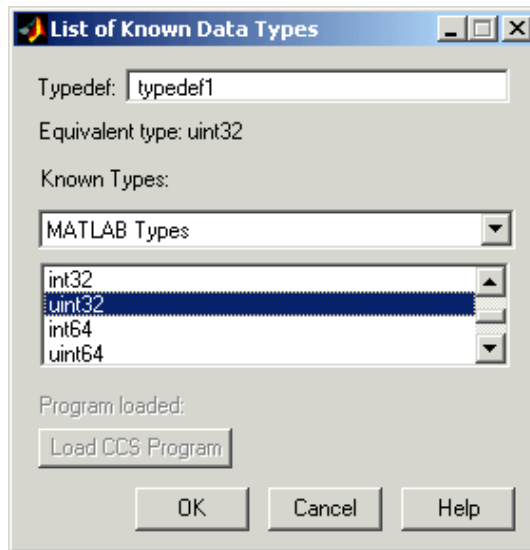
Now start the DTM with the `cc` object. So far you have not loaded a file on the processor.

```
datatypemanager(cc);
```

With the DTM open, you can create a few custom data types.

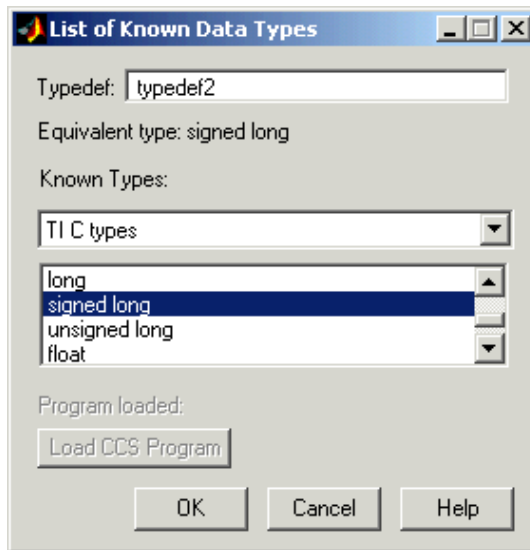
First Example

Create a typedef (typedef1) that uses a MATLAB software data type. typedef1 uses the equivalent data type uint32.



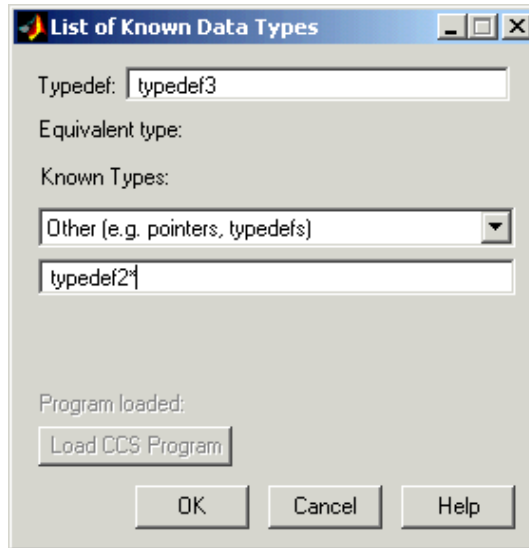
Second Example

Create a second typedef (typedef2) that uses one of the TI C data types. Define typedef2 to use the signed long data type.



Third Example

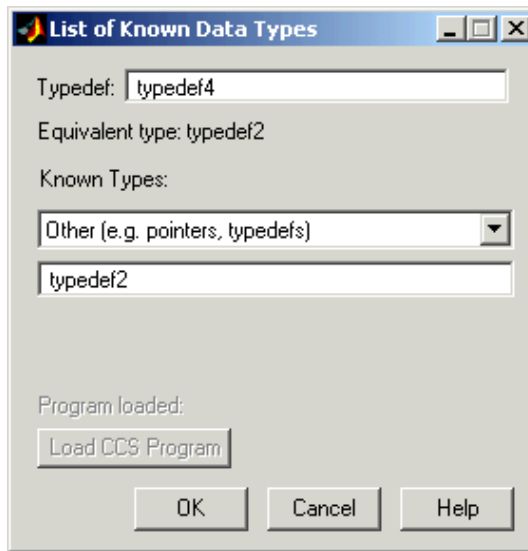
Create a typedef (`typedef3`) that refers to another typedef (`typedef2`). Call this a nested typedef.



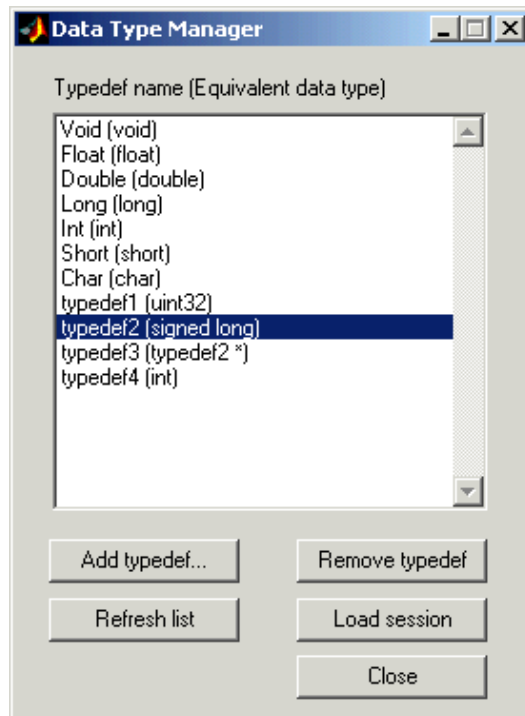
Notice that the referenced typedef, `typedef2`, is entered as a pointer (indicated by the added asterisk). Using the pointer form lets MATLAB software recognize the data type that `typedef2` represents. If you do not use the pointer, MATLAB software converts `typedef3` to a default value equivalent data type, in this case, `int`.

datatypemanager

The next figure shows `typedef4` created to use `typedef2` rather than `typedef2*` for a nested typedef. Under **Equivalent type**, `typedef4` has an equivalent data type of `typedef2`, as specified. But, when you look at the list of known data types in the Data Type Manager dialog box, you see that `typedef4` maps to `int`, not `typedef2`, or eventually signed long.



Here is the DTM after you create all the example custom data types. Take note of `typedef4` in this listing. You see `typedef4` defaults to an equivalent data type `int`, where `typedef3`, also a nested type definition, retains the equivalent data type you assigned. Now you are ready to use a function that includes your custom type definitions in your hardware-in-the-loop development work.



dir

Purpose (For CCS) List files in current CCS IDE working directory

Syntax `dir(cc)`

Description `dir(cc)` lists the files and directories in the current CCS IDE working directory. This does not reflect your MATLAB software working directory or change the working directory.

Use `cd` to change your CCS IDE working directory.

See Also `cd`, `open`

Purpose

(For CCS) Disable RTDX interface, specified channel, or all RTDX channels

Note Support for `disable` on C5000 and C6000 processors will be removed in a future version.

Syntax

```
disable(rx, 'channel')
disable(rx, 'all')
disable(rx)
```

Description

`disable(rx, 'channel')` disables the open channel specified by the string *channel*, for *rx*. Input argument *rx* represents the RTDX portion of the associated link to CCS IDE.

`disable(rx, 'all')` disables all the open channels associated with *rx*.

`disable(rx)` disables the RTDX interface for *rx*.

Important Requirements for Using `disable`

On the processor side, `disable` depends on RTDX to disable channels or the interface. You must meet the following requirements to use `disable`:

- 1 The processor must be running a program.
- 2 You enabled the RTDX interface.
- 3 Your processor program polls periodically.

Examples

When you have opened and used channels to communicate with a processor, you should disable the channels and RTDX before ending your session. Use `disable` to switch off open channels and disable RTDX, as follows:

```
disable(cc.rtdx, 'all') % Disable all open RTDX channels.
disable(cc.rtdx)       % Disable RTDX interface.
```

disable

See Also

close, enable, open

Purpose (For CCS) Display properties of object that refers to CCS IDE or RTDX link

Note `display(rx)` produces a warning on C5000 and C6000 processors and will be removed in a future version.

Syntax

```
display(cc)
display(rx)
display(objectname)
display(cc.type)
```

Description This function is similar to omitting the closing semicolon from an expression on the command line, except that `display` does not display the variable name. `display` provides a formatted list of the property names and property values for a `ticcs` object. To return the configuration data, `display` calls the function `disp`. To return a list of object properties, listed by the actual property names, use `get` with the object.

`display(cc)` returns the information about the `cc` object, listing the properties and values assigned to `cc`.

`display(rx)` returns the information about the `rtdx` object that is part of a `cc` object, listing the properties and values assigned to `cc.rtdx`.

`display(objectname)` returns the properties and property values for `objectname`. This syntax supports all objects except `cc`, `rtdx`, and `cc.type`.

`display(cc.type)` returns the properties and property values for the `cc.type` object. Note that the properties associate with the `cc` object.

The following example illustrates the default display for a link to CCS IDE:

```
cc = ticcs;
```

display

```
display(cc)
TICCS Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0
```

Using display with Multiprocessor Hardware

To support boards that contain more than one processor, `display` behaves slightly differently when `cc` accesses multiprocessor boards.

The syntax

```
display(cc)
```

returns information about all of the members of the object. When the processor has multiple processors, the information returned includes the details of all of the available processors on the processor.

Examples

Try this example to see the display for an RTDX link to a processor:

```
cc = ticcs;
rx=(cc.rtdx)    % Assign the RTDX portion of cc to rx.

RTDX channels    : 0

display(rx)

RTDX channels    : 0
```

Purpose

(For CCS) Enable RTDX interface, specified channel, or all RTDX channels

Note Support for `enable` on C5000 and C6000 processors will be removed in a future version.

Syntax

```
enable(rx, 'channel')
enable(rx, 'all')
enable(rx)
```

Description

`enable(rx, 'channel')` enables the open channel specified by the string `channel`, for RTDX link `rx`. The input argument `rx` represents the RTDX portion of the associated link to CCS IDE.

`enable(rx, 'all')` enables all the open channels associated with `rx`.

`enable(rx)` enables the RTDX interface for `rx`.

Important Requirements for Using `enable`

On the processor side, `enable` depends on RTDX to enable channels. You must meet the following requirements to use `enable`:

- 1** The processor must be running a program when you enable the RTDX interface. When the processor is not running, the state defaults to disabled.
- 2** You must enable the RTDX interface before you enable individual channels.
- 3** Channels must be open.
- 4** Your processor program must poll periodically.
- 5** Using code in the program running on the processor to enable channels overrides the default disabled state of the channels.

enable

Examples

To use channels to RTDX, you must both open and enable the channels:

```
cc = ticcs; % Create a new connection to the IDE.  
enable(cc.rtdx) % Enable the RTDX interface.  
open(cc.rtdx,'inputchannel','w') % Open a channel for sending  
                                % data to the processor.  
enable(cc.rtdx,'inputchannel') % Enable the channel so you can use  
                                % it.
```

See Also

disable, open

Purpose (For CCS) Flush data or messages from specified RTDX channels

Note flush support C5000 and C6000 processors will be removed in a future version.

Syntax

```
flush(rx,channel,num,timeout)
flush(rx,channel,num)
flush(rx,channel,[],timeout)
flush(rx,channel)
flush(rx,'all')
```

Description `flush(rx,channel,num,timeout)` removes *num* oldest data messages from the RTDX channel queue specified by *channel* in *rx*. To determine how long to wait for the function to complete, `flush` uses *timeout* (in seconds) rather than the global timeout period stored in *rx*. `flush` applies the timeout processing when it flushes the last message in the channel queue, because the flush function performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,channel,num)` removes the *num* oldest messages from the RTDX channel queue in *rx* specified by the string *channel*. `flush` uses the global timeout period stored in *rx* to determine how long to wait for the process to complete. Compare this to the previous syntax that specifies the timeout period. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,channel,[],timeout)` removes all data messages from the RTDX channel queue specified by *channel* in *rx*. To determine how long to wait for the function to complete, `flush` uses *timeout* (in seconds) rather than the global timeout period stored in *rx*. `flush` applies the timeout processing when it flushes the last message in the channel queue, because `flush` performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

flush

`flush(rx,channel)` removes all pending data messages from the RTDX channel queue specified by *channel* in *rx*. Unlike the preceding syntax options, you use this statement to remove messages for both read-configured and write-configured channels.

`flush(rx,'all')` removes all data messages from all RTDX channel queues.

When you use `flush` with a write-configured RTDX channel, Embedded IDE Link sends all the messages in the write queue to the processor. For read-configured channels, `flush` removes one or more messages from the queue depending on the input argument *num* you supply and disposes of them.

Examples

To demonstrate `flush`, this example writes data to the processor over the input channel, then uses `flush` to remove a message from the read queue for the output channel:

```
cc = ticcs;
rx = cc.rtdx;
open(rx,'ichan','w');
enable(rx,'ichan');
open(rx,'ochan','r');
enable(rx,'ochan');
indata = 1:10;
writemsg(rx,'ichan',int16(indata));
flush(rx,'ochan',1);
```

Now flush the remaining messages from the read channel:

```
flush(rx,'ochan','all');
```

See Also

`enable`, `open`

Purpose	(For CCS) Terminate execution of process running on processor
Syntax	<code>halt(cc, timeout)</code> <code>halt(cc)</code>
Description	<p><code>halt(cc, timeout)</code> immediately stops program execution by the processor. After the processor stops, <code>halt</code> returns to the host. <i>timeout</i> defines, in seconds, how long the host waits for the processor to stop running. To resume processing after you halt the processor, use <code>run</code>. Also, the <code>read(cc, 'pc')</code> function can determine the memory address where the processor stopped after you use <code>halt</code>.</p> <p><i>timeout</i> defines the maximum time the routine waits for the processor to stop. If the processor does not stop within the specified timeout period, the routine returns with a timeout error.</p> <p><code>halt(cc)</code> immediately stops program execution by the processor. After the processor stops, <code>halt</code> returns to the host. In this syntax, the timeout period defaults to the global timeout period specified in <code>cc</code>. Use <code>get(cc)</code> to determine the global timeout period.</p>

Using halt with Multiprocessor Boards

When you issue a `halt` from the command line, it applies to every processor that the `cc` object represents. Thus `halt` stops every running processor for the object.

Examples

Use one of the provided demonstration programs to show how `halt` works. From the CCS IDE demonstration programs, load and run `volume.out`.

At the MATLAB software prompt create a link to CCS IDE

```
cc = ticcs
```

Check whether the program `volume.out` is running on the processor.

```
isrunning(cc)
```

halt

```
ans =  
    1  
cc.isrunning % Alternate syntax for checking the run status.  
  
ans =  
    1  
halt(cc) % Stop the running application on the processor.  
isrunning(cc)  
  
ans =  
    0
```

Issuing the halt stopped the process on the processor. Checking in CCS IDE shows that the process has stopped.

See Also

ticcs, isrunning, run

Purpose (For CCS) Information about processor

Note Support for `info(rx)` on C5000 and C6000 processors will be removed in a future version.

Syntax

```
info = info(cc)
info = info(rx)
```

Description `info = info(cc)` returns the property names and property values associated with the processor accessed by `cc`. `info` is a structure containing the following information elements and values:

Structure Element	Data Type	Description
<code>info.procname</code>	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with <code>cc</code> .
<code>info.isbigendian</code>	Boolean	Value describing the byte ordering used by the processor. When the processor is big-endian, this value is 1. Little-endian processors return 0.
<code>info.family</code>	Integer	Three-digit integer that identifies the processor family, ranging from 000 to 999. For example, 320 for Texas Instruments digital signal processors.

Structure Element	Data Type	Description
<code>info.subfamily</code>	Decimal	Decimal representation of the hexadecimal identification value that TI assigns to the processor to identify the processor subfamily. IDs range from 0x000 to 0x3822. Use <code>dec2hex</code> to convert the value in <code>info.subfamily</code> to standard notation. For example <code>dec2hex(info.subfamily)</code> produces '67' when the processor is a member of the 67xx processor family.
<code>info.timeout</code>	Integer	Default timeout value MATLAB software uses when transferring data to and from CCS. All functions that use a timeout value have an optional <i>timeout</i> input argument. When you omit the optional argument, MATLAB software uses this default value – 10s.

`info = info(rx)` returns `info` as a cell array containing the names of your open RTDX channels.

Using info with Multiprocessor Boards

Method `info` works with processors that have more than one processor by returning the information for each processor accessed by the `cc` object you created with `ticcs`. The structure of information returned is identical to the single processor case, for every included processor.

Examples

On a PC with a simulator configured in CCS IDE, `info` returns the configuration for the processor being simulated:

```
info(cc)

ans =

    procname: 'CPU'
    isbigendian: 0
```

```
family: 320
subfamily: 103
timeout: 10
```

This example simulates the TMS320C6211 processor running in little-endian mode. When you use CCS Setup Utility to change the processor from little-endian to big-endian, `info` shows the change.

```
info(cc)

ans =

    procname: 'CPU'
  isbigendian: 1
    family: 320
  subfamily: 103
    timeout: 10
```

If you have two open channels, `chan1` and `chan2`,

```
info = info(rx)

returns

info =
'chan1'
'chan2'
```

where `info` is a cell array. You can dereference the entries in `info` to manipulate the channels. For example, you can close a channel by dereferencing the channel in `info` in the `close` function syntax.

```
close(rx.info{1,1})
```

See Also

`ticcs`, `dec2hex`

insert

Purpose (For CCS) Add debug point to source file or address in CCS

Syntax

```
insert(cc,addr,'type')
insert(cc,addr,'type',timeout)
insert(cc,length)
insert(cc,filename,line,'type')
insert(cc,filename,line,'type',timeout)
insert(cc,filename,line)
```

Description `insert(cc,addr,'type')` adds a debug point located at the memory address identified by `addr` for your processor digital signal processor. The link `cc` identifies which processor has the debug point to insert. CCS provides several types of debug points specified by `type`. Options for `type` include the following strings to define Breakpoints, Probe Points, and Profile points:

- 'break' — add a breakpoint. It defines a point at which program execution stops.
- '' — same as 'break'.
- 'probe' — add a Probe Point that updates a CCS window during program execution. When CCS connects your probe point to a window, the window gets updated only when the executing program reaches the Probe Point.
- 'profile' — add a point in an executing program at which CCS gathers statistics about events that occurred after encountering the previous profile point, or from the start of your program.

When you use it, `insert` operates in *blocking* mode, meaning that after you issue the `insert` command, you do not regain control in the MATLAB environment until the `insert` breakpoint operation is completed successfully—you are blocked from further processing. `insert` waits for the period defined by either `timeout` or `cc.timeout`. If the `insert` operation does not get completed within the specified time period, `insert` returns an error and control.

When you use the *line* input argument to insert a breakpoint on a specified line, *line* must represent a valid line. If *line* does not specify a valid line, insert returns an error and does not insert the breakpoint.

Enter *addr* as a hexadecimal address, not as a ANSI C function name, valid ANSI C expression, or a symbol name.

To learn more about the behavior of the various debugging points refer to your CCS documentation.

`insert(cc,addr,'type',timeout)` adds the optional input parameter *timeout* that determines how long Embedded IDE Link waits for a response to a request to insert a breakpoint. If the response is not received before the timeout period expires, the insertion process fails with a timeout error. Adding the *timeout* input argument is valid only when you are inserting a breakpoint. When you omit the *timeout* argument, insert uses the default value defined by `cc.timeout`

`insert(cc,length)` is the same as the previous syntax except the *type* string defaults to 'break' for inserting a Breakpoint.

`insert(cc,filename,line,'type')` lets you specify the line where you are inserting the debug point. *line*, in decimal notation, specifies the line number in *filename* in CCS where you are adding the debug point.

To identify the source file, *filename* contains the name of the file in CCS, entered as a string in single quotation marks. Do not include the path to the file. insert ignores the file path information if you add it to *filename.type* accepts one of three strings—`break`, `probe`, or `profile`—as defined previously.

When the line or file you specified does not exist, Embedded IDE Link returns an error explaining that it could not insert the debug point.

`insert(cc,filename,line,'type',timeout)` adds the optional input parameter *timeout* that determines how long Embedded IDE Link waits for a response to a request to insert a breakpoint. If the response is not received before the timeout period expires, the insertion process fails with a timeout error. Adding the *timeout* input argument is valid only when you are inserting a breakpoint. When you omit the *timeout*

`insert(cc, filename, line)` defaults to type 'break' to insert a breakpoint.

Example

Open a project in CCS IDE, such as `volume.pjt` in the tutorial folder where you installed CCS IDE. Use Embedded IDE Link functions to open the project and activate the appropriate source file where you add the breakpoint. Remember to load the program file `volume.out` so you can access symbols and their addresses.

```
cd (cc, 'c:\ti\tutorial\sim62xx\volume1') % Default install;
wd=cd(cc);

wd =

c:\ti\tutorial\sim62xx\volume1

open(cc, 'volume.pjt');

build(cc, 30);
```

Now add a breakpoint and a probe point.

```
insert(cc, 15424, 'break') % Adds a breakpoint at symbol "main"
insert(cc, 'volume.c', 47, 'probe') % Adds a probe point on line 47
```

Switch to CCS IDE and open `volume.c`. Note the blue diamond and red circle in the left margin of the `volume.c` listing. Red circles indicate Breakpoints and blue diamonds indicate Probe Points.

Use `symbol` to return a structure listing the symbols and their addresses for the current program file. `symbol` returns a structure that contains all the symbols. To display all the symbols with addresses, use a loop construct like the following:

```
for k=1:length(s), disp(k), disp(s(k)), end
```

where structure `s` holds the symbols and addresses.

See Also address, remove, run

isenabled

Purpose

(For CCS) Determine whether RTDX link is enabled for communications

Note Support for `isenabled` on C5000 and C6000 processors will be removed in a future version.

Syntax

```
isenabled(rx, 'channel')  
isenabled(rx)
```

Description

`isenabled(rx, 'channel')` returns `ans=1` when the RTDX channel specified by string `'channel'` is enabled for read or write communications. When `'channel'` has not been enabled, `isenabled` returns `ans=0`.

`isenabled(rx)` returns `ans=1` when RTDX has been enabled, independent of any channel. When you have not enabled RTDX you get `ans=0` back.

Important Requirements for Using `isenabled`

On the processor side, `isenabled` depends on RTDX to determine and report the RTDX status. Therefore the you must meet the following requirements to use `isenabled`.

- 1** The processor must be running a program when you query the RTDX interface.
- 2** You must enable the RTDX interface before you check the status of individual channels or the interface.
- 3** Your processor program must be polling periodically for `isenabled` to work.

Note For `isenabled` to return reliable results, your processor must be running a loaded program. When the processor is not running, `isenabled` returns a status that may not represent the true state of the channels or RTDX.

Examples

With a program loaded on your processor, you can determine whether RTDX channels are ready for use. Restart your program to be sure it is running. The processor must be running for `isEnabled` to work, as well as for `enable` to work. This example creates a `ticcs` object `cc` to begin.

```
cc.restart
cc.run('run');
cc.rtdx.enable('ichan');
cc.rtdx.isEnabled('ichan')
```

MATLAB software returns 1 indicating that your channel 'ichan' is enabled for RTDX communications. To determine the mode for the channel, use `cc.rtdx` to display the properties of object `cc.rtdx`.

See Also

`clear`, `disable`, `enable`

isreadable

Purpose (For CCS) Determine whether MATLAB software can read specified memory block

Note Support for `isreadable(rx, 'channel')` on C5000 and C6000 processors will be removed in a future version.

Syntax

```
isreadable(cc,address,'datatype',count)
isreadable(cc,address,'datatype')
isreadable(rx,'channel')
```

Description `isreadable(cc,address,'datatype',count)` returns 1 if the processor referred to by `cc` can read the memory block defined by the `address`, `count`, and `datatype` input arguments. When the processor cannot read any portion of the specified memory block, `isreadable` returns 0. You use the same memory block specification for this function as you use for the read function.

The data block being tested begins at the memory location defined by `address`. `count` determines the number of values to be read. `datatype` defines the format of data stored in the memory block. `isreadable` uses the `datatype` string to determine the number of bytes to read per stored value. For details about each input parameter, read the following descriptions.

`address` — `isreadable` uses `address` to define the beginning of the memory block to read. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector `[location, page]`, a string, or a decimal value.

When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, `ticcs` sets the page to 0 at creation if you omit the page property as an input argument. For processors that have one memory

page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
'1F'	String	Location is 31 decimal on the page referred to by <code>cc.page</code>
10	Decimal	Address is 10 decimal on the page referred to by <code>cc.page</code>
[18,1]	Vector	Address location 10 decimal on memory page 1 (<code>cc.page = 1</code>)

To specify the address in hexadecimal format, enter the *address* property value as a string. `isreadable` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. Note that when you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc.page`.

count — a numeric scalar or vector that defines the number of *datatype* values to test for being readable. To assure parallel structure with `read`, *count* can be a vector to define multidimensional data blocks. This function always tests a block of data whose size is the product of the dimensions of the input vector.

datatype — a string that represents a MATLAB software data type. The total memory block size is derived from the value of *count* and the *datatype* you specify. *datatype* determines how many bytes to check for each memory value. `isreadable` supports the following data types:

isreadable

<i>datatype</i> String	Number of Bytes/Value	Description
'double'	8	Double-precision floating point values
'int8'	1	Signed 8-bit integers
'int16'	2	Signed 16-bit integers
'int32'	4	Signed 32-bit integers
'single'	4	Single-precision floating point data
'uint8'	1	Unsigned 8-bit integers
'uint16'	2	Unsigned 16-bit integers
'uint32'	4	Unsigned 32-bit integers

Like the `iswritable`, `write`, and `read` functions, `isreadable` checks for valid address values. Illegal address values would be any address space larger than the available space for the processor – 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

`isreadable(cc, address, 'datatype')` returns 1 if the processor referred to by `cc` can read the memory block defined by the `address`, and `datatype` input arguments. When the processor cannot read any portion of the specified memory block, `isreadable` returns 0. Notice that you use the same memory block specification for this function as you use for the `read` function. The data block being tested begins at the memory location defined by `address`. When you omit the `count` option, `count` defaults to one.

`isreadable(rx, 'channel')` returns a 1 when the RTDX channel specified by the string `channel`, associated with link `rx`, is configured for read operation. When `channel` is not configured for reading, `isreadable` returns 0.

Like the `iswritable`, `read`, and `write` functions, `isreadable` checks for valid address values. Illegal address values are address spaces larger than the available space for the processor – 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

Note `isreadable` relies on the memory map option in CCS IDE. If you did not properly define the memory map for the processor in CCS IDE, `isreadable` does not produce useful results. Refer to your Texas Instruments' Code Composer Studio documentation for information on configuring memory maps.

Examples

When you write scripts to run models in the MATLAB environment and CCS IDE, the `isreadable` function is very useful. Use `isreadable` to check that the channel from which you are reading is configured properly.

```
cc = ticcs;
rx = cc.rtdx;

% Define read and write channels to the processor linked by cc.
open(rx, 'ichannel', 'r');
open(rx, 'ochannel', 'w');
enable(rx, 'ochannel');
enable(rx, 'ichannel');

isreadable(rx, 'ochannel')
ans=
    0
isreadable(rx, 'ichannel')
ans=
    1
```

isreadable

Now that your script knows that it can read from `ichannel`, it proceeds to read messages as required.

See Also

`hex2dec`, `iswritable`, `read`

Purpose (For CCS) Determine whether processor supports RTDX

Note Support for `isrtdxcapable` on C5000 and C6000 processors will be removed in a future version.

Syntax `b=isrtdxcapable(cc)`

Description `b=isrtdxcapable(cc)` returns `b=1` when the processor referenced by object `cc` supports RTDX. When the processor does not support RTDX, `isrtdxcapable` returns `b=0`.

Using `isrtdxcapable` with Multiprocessor Boards

When your board contains more than one processor, `isrtdxcapable` checks each processor on the processor, as defined by the `cc` object, and returns the RTDX capability for each processor on the board. In the returned variable `b`, you find a vector that contains the information for each accessed processor.

Examples Create a link to your C6711 DSK. Test to see if the processor on the board supports RTDX. It should.

```
cc=ticcs; %Assumes you have one board and it is the C6711 DSK.
b=isrtdxcapable(cc)
b =
    1
```

isrunning

Purpose (For CCS) Determine whether processor is executing process

Syntax `isrunning(cc)`

Description `isrunning(cc)` returns 1 when the processor is executing a program. When the processor is halted, `isrunning` returns 0.

Using isrunning with Multiprocessor Boards

When your board contains more than one processor, `isrunning` checks each processor on the processor, as defined by the `cc` object, and returns the state for each processor on the board. In the returned variable `b`, you find a vector that contains the information for each accessed processor.

By providing a return variable, as shown here,

```
b = isrunning(cc)
```

`b` contains a vector that holds the information about the state of all processors accessed by `cc`.

Examples

`isrunning` lets you determine whether the processor is running. After you load a program to the processor, use `isrunning` to be sure the program is running before you enable RTDX channels.

```
cc = ticcs;

isrunning(cc)

ans =

    0
% Load a program to the processor.

run(cc)
isrunning(cc)

ans =
```



```
1  
halt(cc)  
isrunning(cc)  
ans =  
0
```

See Also halt, restart, run

isvisible

Purpose (For CCS) Determine whether CCS IDE is running

Syntax `isvisible(cc)`

Description `isvisible(cc)` determines whether CCS IDE is running on the desktop and the window is open. If CCS IDE window is open, `isvisible` returns 1. Otherwise, the result is 0 indicating that CCS IDE is either not running or is running in the background.

Examples Test to see if CCS IDE is running. Start CCS IDE. Then open MATLAB software. At the prompt, enter

```
cc=ticcs
```

```
TICCS Object:
```

```
API version      = 1.0
Processor type   = C67
Processor name   = CPU
Running?        = No
Board number     = 0
Processor number = 0
Default timeout  = 10.00 secs
```

```
RTDX Object:
```

```
Timeout: 10.00 secs
Number of open channels: 0
```

MATLAB software creates a link to CCS IDE and leaves CCS IDE visible on your desktop.

```
isvisible(cc)
```

```
ans =
```

```
1
```

Now, change the visibility state to 0, or invisible, and check the state.

```
visible(cc,0)
isvisible(cc)

ans =

     0
```

Notice that CCS IDE is not visible on your desktop. Recall that MATLAB software did not open CCS IDE. When you close MATLAB software with CCS IDE in this invisible state, CCS IDE remains running in the background. To close it, do one of the following.

- Open MATLAB software. Create a new link to CCS IDE. Use the new link to make CCS IDE visible. Close CCS IDE.
- Open Microsoft Windows® Task Manager. Click **Processes**. Find and highlight `cc_app.exe`. Click **End Task**.

See Also

`info`, `visible`

iswritable

Purpose

(For CCS) Determine whether MATLAB software can write to specified memory block

Note Support for `iswritable(rx, 'channel')` on C5000 and C6000 processors will be removed in a future version.

Syntax

```
iswritable(cc, address, 'datatype', count)
iswritable(cc, address, 'datatype')
iswritable(rx, 'channel')
```

Description

`iswritable(cc, address, 'datatype', count)` returns 1 if MATLAB software can write to the memory block defined by the `address`, `count`, and `datatype` input arguments on the processor referred to by `cc`. When the processor cannot write to any portion of the specified memory block, `iswritable` returns 0. You use the same memory block specification for this function as you use for the `write` function.

The data block being tested begins at the memory location defined by `address`. `count` determines the number of values to write. `datatype` defines the format of data stored in the memory block. `iswritable` uses the `datatype` parameter to determine the number of bytes to write per stored value. For details about each input parameter, read the following descriptions.

address — `iswritable` uses `address` to define the beginning of the memory block to write to. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector `[location, page]`, a string, or a decimal value. When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, `ticcs` sets the page to 0 at creation if you omit the page property as an input argument.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
1F	String	Location is 31 decimal on the page referred to by <code>cc.page</code>
10	Decimal	Address is 10 decimal on the page referred to by <code>cc.page</code>
[18,1]	Vector	Address location 10 decimal on memory page 1 (<code>cc.page = 1</code>)

To specify the address in hexadecimal format, enter the address property value as a string. `iswritable` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. Note that when you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc.page`.

`count` — a numeric scalar or vector that defines the number of `datatype` values to test for being writable. To assure parallel structure with `write`, `count` can be a vector to define multidimensional data blocks. This function always tests a block of data whose size is the total number of elements in matrix specified by the input vector. If `count` is the vector [10 10 10]

```
iswritable(cc,31,[10 10 10])
```

`iswritable` writes 1000 values (10*10*10) to the processor. For a two-dimensional matrix defined with `count` as

```
iswritable(cc,31,[5 6])
```

iswritable

`iswritable` writes 30 values to the processor.

`datatype` — a string that represents a MATLAB data type. The total memory block size is derived from the value of `count` and the specified `datatype`. `datatype` determines how many bytes to check for each memory value. `iswritable` supports the following data types:

datatype String	Description
'double'	Double-precision floating point values
'int8'	Signed 8-bit integers
'int16'	Signed 16-bit integers
'int32'	Signed 32-bit integers
'single'	Single-precision floating point data
'uint8'	Unsigned 8-bit integers
'uint16'	Unsigned 16-bit integers
'uint32'	Unsigned 32-bit integers

`iswritable(cc,address,'datatype')` returns 1 if the processor referred to by `cc` can write to the memory block defined by the `address`, and `count` input arguments. When the processor cannot write any portion of the specified memory block, `iswritable` returns 0. Notice that you use the same memory block specification for this function as you use for the `write` function. The data block tested begins at the memory location defined by `address`. When you omit the `count` option, `count` defaults to one.

Note `iswritable` relies on the memory map option in CCS IDE. If you did not properly define the memory map for the processor in CCS IDE, this function does not produce useful results. Refer to your Texas Instruments' Code Composer Studio documentation for information on configuring memory maps.

Like the `isreadable`, `read`, and `write` functions, `iswritable` checks for valid address values. Illegal address values would be any address space larger than the available space for the processor – 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

`iswritable(rx, 'channel')` returns a Boolean value signifying whether the RTDX channel specified by `channel` and `rx`, is configured for write operations.

Examples

When you write scripts to run models in MATLAB software and CCS IDE, the `iswritable` function is very useful. Use `iswritable` to check that the channel to which you are writing to is indeed configured properly.

```
cc = ticcs;
rx = cc.rtdx;

% Define read and write channels to the processor linked by cc.
open(rx, 'ichannel', 'r');
open(rx, 'ochannel', 'w');
enable(rx, 'ochannel');
enable(rx, 'ichannel');

iswritable(rx, 'ochannel')
ans=
    1
iswritable(rx, 'ichannel')
ans=
    0
```

Now that your script knows that it can write to 'ichanne'l, it proceeds to write messages as required.

See Also

`hex2dec`, `isreadable`, `read`

list

Purpose (For CCS) Information listings from CCS

Syntax

```
list(ff,varname)
infolist = list(cc,'type')
infolist = list(cc,'type',typename)
```

Note `list(cc,type)` produces an error.

Description `list(ff,varname)` lists the local variables associated with the function accessed by function object `ff`. Compare to `list(cc,'variable','varname')` which works the same way to return variables from `ticcs` object `cc`.

Note `list` does not recognize or return information about variables that you declare in your code but that are not used or initialized.

Some restrictions apply when you use `list` with function objects. `list` generates an error in the following circumstances:

- When `varname` is not a valid input argument for the function accessed by `ff`

For example, if your function declaration is

```
int foo(int a)
```

but you request information about input argument `b`, which is not defined

```
list(ff,'b')
```

MATLAB software returns an error.

- When `varname` is the same as a variable assigned by MATLAB software. Usually this happens when you use `declare` to pass

a function declaration to MATLAB software and the declaration string does not match the declaration for `ff` as determined when you created `ff`.

In an example that demonstrates this problem, the function declaration has a name for the first input, `a`. In the `declare` call, the declaration string does not provide a name for the first input, just a data type, `int`. When you issue the `declare` call, MATLAB software names the first input `ML_Input1`. If you try to use `list` to get information about the input named `ML_Input`, `list` returns an error. Here is the code, starting with the function declaration in your code:

```
int foo(int a) % Function declaration in your source code
declare(ff,'decl','int foo(int)')
% MATLAB generates a warning that it has assigned the name
% ML_Input to the first input argument
list(ff,'ML_Input') % list returns an error for this call
```

- When `varname` does not match the input name in the function declaration provided in your source code, as compared to the declaration string you used in a `declare` operation.

Assume your source code includes a function declaration for `foo`:

```
int foo(int a);
```

Now pass a declaration for `foo` to MATLAB software:

```
declare(ff,'decl','int foo(int b)')
```

MATLAB software issues a warning that the input names do not match. When you use `list` on the input argument `b`,

```
list(ff,'b')
```

`list` returns an error.

- When `varname` is an input to a library function. `list` always fails in this case. It does not matter whether you use `declare` to provide the declaration string for the library function.

Note When you call `list` for a variable in a function object `list(ff, varname)` the `address` field may contain an incorrect address if the program counter is not within the scope of the function that includes `varname` when you call `list`.

`infolist = list(cc, type)` reads information about your CCS session and returns it in `infolist`. Different types of information and return formats apply depending on the input arguments you supply to the `list` function call. The `type` argument specifies which information listing to return. To determine the information that `list` returns, use one of the following as the `type` parameter string:

- **project** — Tell `list` to return information about the current project in CCS.
- **variable** — Tell `list` to return information about one or more embedded variables.
- **globalvar** — Tell `list` to return information about one or more global embedded variables.
- **function** — Tell `list` to return details about one or more functions in your project.
- **type** — Tell `list` to return information about one or more defined data types, including `struct`, `enum`, and `union`. ANSI C data type typedefs are excluded from the list of data types.

Note, the `list` function returns dynamic CCS information that can be altered by the user. Returned listings represent snapshots of the current CCS configuration only. Be aware that earlier copies of `infolist` might contain stale information.

Also, `list` may report incorrect information when you make changes to variables from MATLAB software. To report variable information, `list` uses the CCS API, which only knows about variables in CCS. Your changes from MATLAB software, such as changing the data type of a variable, do not appear through the API and `list`. For example, the following operations return incorrect or old data information from `list`.

Suppose your original prototype is

```
unsigned short tgtFunction7(signed short signedShortArray1[]);
```

After creating the function object `fcnObj`, perform a `declare` operation with this string to change the declaration:

```
unsigned short tgtFunction7(unsigned short signedShortArray1[]);
```

Now try using `list` to return information about `signedShortArray1`.

```
list(fcnObj, 'signedShortArray1')  
  
address: [3442 1]  
location: [1x66 char]  
size: 1  
type: 'short *'  
bitsize: 16  
reftype: 'short'  
referent: [1x1 struct]  
member_pts_to_same_struct: 0  
name: 'signedShortArray1'
```

The `type` field reports the original data type `short`.

You get this is because `list` uses the CCS API to query information about any particular variable. As far as the API is concerned, the first input variable is a `short*`. Changing the declaration does not change anything.

list

`infolist = list(cc, 'project')` returns a vector of structures containing project information in the format shown here when you specify option type as **project**.

infolist Structure Element	Description
<code>infolist(1).name</code>	Project file name (with path).
<code>infolist(1).type</code>	Project type — <code>project</code> , <code>projlib</code> , or <code>projext</code> , refer to <code>new</code>
<code>infolist(1).proccesortype</code>	String description of processor CPU
<code>infolist(1).srcfiles</code>	Vector of structures that describes project source files. Each structure contains the name and path for each source file — <code>infolist(1).srcfiles.name</code>
<code>infolist(1).buildcfg</code>	Vector of structures that describe build configurations, each with the following entries: <ul style="list-style-type: none">• <code>infolist(1).buildcfg.name</code> — the build configuration name• <code>infolist(1).buildcfg.outpath</code> — the default directory for storing the build output.
<code>infolist(2)....</code>	...
<code>infolist(n)....</code>	...

`infolist = list(cc, 'variable')` returns a structure of structures that contains information on all local variables within scope. The list also includes information on all global variables. Note, however, that if a local variable has the same symbol name as a global variable, `list` returns the information about the local variable.

`infolist = list(cc, 'variable', varname)` returns information about the specified variable `varname`.

`infolist = list(cc, 'variable', varnamelist)` returns information about variables in a list specified by `varnamelist`. The information returned in each structure follows the format below when you specify option type as **variable**:

infolist Structure Element	Description
<code>infolist.varname(1).name</code>	Symbol name
<code>infolist.varname(1).isglobal</code>	Indicates whether symbol is global or local
<code>infolist.varname(1).location</code>	Information about the location of the symbol
<code>infolist.varname(1).size</code>	Size per dimension
<code>infolist.varname(1).uclass</code>	ticcs object class that matches the type of this symbol
<code>infolist.varname(1).bitsize</code>	Size in bits. More information is added to the structure depending on the symbol type.
<code>infolist.(varname1).type</code>	data type of symbol
<code>infolist.varname(2)....</code>	...
<code>infolist.varname(n)....</code>	...

`list` uses the variable name as the field name to refer to the structure information for the variable.

`infolist = list(cc, 'globalvar')` returns a structure that contains information on all global variables.

`infolist = list(cc, 'globalvar', varname)` returns a structure that contains information on the specified global variable.

`infolist = list(cc, 'globalvar', varnamelist)` returns a structure that contains information on global variables in the list. The

list

returned information follows the same format as the syntax
`infolist = list(cc, 'variable', ...)`.

`infolist = list(cc, 'function')` returns a structure that contains information on all functions in the embedded program.

`infolist = list(cc, 'function', functionname)` returns a structure that contains information on the specified function `functionname`.

`infolist = list(cc, 'function', functionnamelist)` returns a structure that contains information on the specified functions in `functionnamelist`. The returned information follows the format below when you specify option type as **function**:

infolist Structure Element	Description
<code>infolist.functionname(1).name</code>	Function name
<code>infolist.functionname(1).filename</code>	Name of file where function is defined
<code>infolist.functionname(1).address</code>	Relevant address information such as start address and end address
<code>infolist.functionname(1).funcvar</code>	Variables local to the function
<code>infolist.functionname(1).uclass</code>	<code>ticcs</code> object class that matches the type of this symbol — function
<code>infolist.functionname(1).funcdecl</code>	Function declaration — where information such as the function return type is contained
<code>infolist.functionname(1).islibfunc</code>	Is this a library function?

infolist Structure Element	Description
<code>infolist.functionname(1).linepos</code>	Start and end line positions of function
<code>infolist.functionname(1).funcinfo</code>	Miscellaneous information about the function
<code>infolist.functionname(2)...</code>	...
<code>infolist.functionname(n)...</code>	...

To refer to the function structure information, `list` uses the function name as the field name.

`infolist = list(cc, 'type')` returns a structure that contains information on all defined data types in the embedded program. This method includes struct, enum and union data types and excludes typedefs. The name of a defined type is its ANSI C struct tag, enum tag or union tag. If the ANSI C tag is not defined, it is referred to by the CCS compiler as '`$faken`' where *n* is an assigned number.

`infolist = list(cc, 'type', typename)` returns a structure that contains information on the specified defined data type.

`infolist = list(cc, 'type', typenamelist)` returns a structure that contains information on the specified defined data types in the list. The returned information follows the format below when you specify option type as **type**:

infolist Structure Element	Description
<code>infolist.typename(1).type</code>	Type name
<code>infolist.typename(1).size</code>	Size of this type
<code>infolist.typename(1).uclass</code>	ticcs object class that matches the type of this symbol. Additional information is added depending on the type

list

infolist Structure Element	Description
infolist.typeName(2)...	...
infolist.typeName(n)...	...

For the field name, `list` uses the type name to refer to the type structure information.

The following list provides important information about variable and field names:

- When a variable name, type name, or function name is not a valid MATLAB software structure field name, `list` replaces or modifies the name so it becomes valid.
- In field names that contain the invalid dollar character \$, `list` replaces the \$ with DOLLAR.
- Changing the MATLAB software field name does not change the name of the embedded symbol or type.

Examples

This first example shows `list` used with a variable, providing information about the variable `varname`. Notice that the invalid field name `_with_underscore` gets changed to `Q_with_underscore`. To make the invalid name valid, `list` inserts the character `Q` before the name.

```
varname1 = '_with_underscore'; % invalid fieldname
list(cc, 'variable', varname1);
ans =

    Q_with_underscore : [varinfo]
ans. Q_with_underscore
ans=

    name: '_with_underscore'
isglobal: 0
location: [1x62 char]
size: 1
```



```

    uclass: 'numeric'
    type: 'int'
    bitsize: 16

```

To demonstrate using `list` with a defined C type, variable `typename1` includes the `type` argument. Because valid field names cannot contain the `$` character, `list` changes the `$` to `DOLLAR`.

```

typename1 = '$fake3'; % name of defined C type with no tag
list(cc,'type',typename1);
ans =

    DOLLARfake0 : [typeinfo]

ans.DOLLARfake0=

    type: 'struct $fake0'
    size: 1
    uclass: 'structure'
    sizeof: 1
    members: [1x1 struct]

```

When you request information about a project in CCS, you see a listing like the following that includes structures containing details about your project.

```

projectinfo=list(cc,'project')

projectinfo =

    name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'
    type: 'project'
    procestype: 'TMS320C67XX'
    srcfiles: [69x1 struct]
    buildcfg: [3x1 struct]

```

See Also

`info`

load

Purpose (For CCS) Transfer program file (*.out, *.obj) to processor in active project

Syntax

```
load(cc,'filename',timeout)
load(cc,'filename')
load(cc,'gelfilename',timeout)
```

Description `load(cc,'filename',timeout)` loads the file specified by `filename` into the processor. `filename` can include a full path to a file, or just the name of a file that resides in the CCS working directory. Use `cd` to check or modify the working directory. Only use `load` with program files that are created by the CCS build process.

`timeout` defines the upper limit on how long MATLAB software waits for the load process to be complete. If this period is exceeded, `load` returns immediately with a timeout error.

`load(cc,'filename')` loads the file specified by `filename` into the processor. `filename` can include a full path to a file, or just the name of a file that resides in the CCS working directory. Use `cd` to check or modify the working directory. Only use `load` with program files that are created by the CCS build process. `timeout` defaults to the global value you set when you created link `cc`.

Note `load` disables all open channels. Open channels revert to disabled.

`load(cc,'gelfilename',timeout)` loads and opens the general extension language (GEL) file named `gelfilename` into CCS, in the active project. `gelfilename` needs to be the full path to the file, or just the file name if the file already shows up in your CCS workspace or project. `load` adds the GEL file to the active project only. To make a different project active so you can add your GEL file to it, use `activate`.

The `timeout` option is not required, as is true for most methods in the product. Using `load` to add a GEL file is identical to using the **File**

> **Load GEL** option in CCS IDE. Your loaded GEL file appears in the GEL files folder in CCS. To remove GEL files, use `remove`. You can load any GEL file — you must be sure the GEL file is the correct one. `load` does not attempt to verify whether the GEL file is appropriate for your hardware or project.

Examples

Taken from the CCS link tutorial, this code prepares for and loads an object file `filename.out` to a processor.

```
projfile = ...
fullfile(matlabroot, 'directoryname', 'directoryname', 'filename')
projpath = fileparts(projfile)
open(cc,projfile) % Open project file
cd(cc,projpath) % Change Code Composer working directory
```

Now use CCS IDE to build your file. Select **Project > Build** from the menu bar in CCS IDE.

With the project build complete, load your `.out` file by entering

```
load(cc, 'filename.out')
```

See Also

`cd`, `dir`, `open`

msgcount

Purpose

(For CCS) Number of messages in read-enabled channel queue

Note Support for msgcount on C5000 and C6000 processors will be removed in a future version.

Syntax

msgcount(rx, 'channel')

Description

msgcount(rx, 'channel') returns the number of unread messages in the read-enabled queue specified by channel for the RTDX interface rx. You cannot use msgcount on channels configured for write access.

Examples

If you have created and loaded a program to the processor, you can write data to the processor, then use msgcount to determine the number of messages in the read queue.

1 Create and load a program to the processor.

2 Write data to the processor from MATLAB software.

```
indata=1:100;  
writemsg(cc.rtdx,'ichannel', int32(indata));
```

3 Use msgcount to determine the number of messages available in the queue.

```
num_of_msgs = msgcount(cc.rtdx,'ichannel')
```

See Also

read, readmat, readmsg

Purpose

(For CCS) Create and open text file, project, or build configuration in CCS IDE

Note `new(cc,objectname,'text')` produces an error.

Syntax

```
new(cc,'objectname','type')
new(cc,'objectname')
```

Description

`new(cc,'objectname','type')` creates and opens an empty object of type named `objectname` in the active project in CCS IDE. The new object can be a text file, a project, or a build configuration. String `objectname` specifies the name of the new object. When you create new text files or projects, `objectname` can include a full path description. When you save your new project or file, CCS IDE stores the file at the processor of the full path.

If you do not provide a full path for your file, `new` stores the file in the CCS IDE working directory when you save it. New files open as active windows in CCS IDE; they are not placed in the active project folders based on their file extension (compare to `add`).

New build configurations always become part of the active project in CCS IDE. Because build configurations always become part of a project, you only need to enter a name to distinguish your new configuration from existing configurations in the project, such as Debug and Release.

To specify the text file or project to create, `objectname` must be the full path name to the file, unless your file is in a directory on your MATLAB software path, or the file is in your CCS working directory. Also, when you create new text files or projects, you must include the file extension in `objectname`.

`type` accepts one of the strings or entries listed in the following table.

type String	Description
'text'	Create a new text file in the active project.

type String	Description
'project'	Create a new project.
'projext'	Create a new CCS external make project. Using this option indicates that your project uses and external makefile. Refer to your CCS documentation for more information about external projects.
'projlib'	Create a new library project with the .lib file extension. Refer to your CCS documentation for more information about library projects.
[]	Create a new project. The [] indicate that you are creating a .pjt file.
'buildcfg'	Create a new build configuration in the active project.

Use `new` to create the file types listed in the following table.

File Types and Extensions Supported by `new` and CCS IDE

File Type to Create	type String Used	Supported Extensions
C/C++ source files	'text'	.c, .cpp, .cc, .ccx, .sa
Assembly source files	'text'	.a*, .s* (excluding .sa, refer to C/C++ source files)
Object and Library files	'text'	.o*, .lib
Linker command file	'text'	.cmd
Project file	'project'	.pjt
Build configuration	'buildcfg'	No extension

Caution After you create an object in CCS IDE, save the file in CCS IDE. `new` does not automatically save the file, and you lose your changes when you close CCS IDE.

`new(cc, 'objectname')` creates a project in CCS IDE, making it the active project. When you omit the `type` option, `new` assumes you are creating a new project and appends the `.pjt` extension to `objectname` to create the project `objectname.pjt`. The `.pjt` extension is the only extension `new` recognizes.

Examples

When you need a new project, create a link to CCS IDE and use the link to make a new project in CCS IDE.

```
cc=ticcs;  
cc.visible(1) % Make CCS IDE visible on your desktop (optional).  
new(cc, 'my_new_project.pjt', 'project');
```

New files of various types result from using `new` to create new active windows in CCS IDE. For instance, make a new ANSI C source file in CCS IDE with the following command:

```
new(cc, 'new_source.c', 'text');
```

In CCS IDE you see your new file as the active window.

See Also

`activate`, `close`, `save`

open

Purpose

(For CCS) Open channel to processor or load file into CCS IDE

Note `open(rx,...)` uses RTDX. Embedded IDE Link no longer supports RTDX for C6000 processors and will remove support for C5000 processors in a future version.

`open(cc,filename,'text')` produces an error.

`open(cc,filename,'workspace')` produces an error.

`open(cc,filename,'program')` produces an error. Use `load` instead.

Syntax

```
open(rx,'channel1','mode1','channel2','mode2',...)  
open(rx,channel,mode)  
open(cc,filename,filetype,timeout)  
open(cc,filename,filetype)  
open(cc,filename)
```

Description

`open(rx,'channel1','mode1','channel2','mode2',...)` opens new RTDX channels associated with the link `rx`. Each new channel uses the string name `channel1`, `channel2`, and so on. For each channel, `open` configures the channel according to the associated mode string. `channel1` uses `mode1`; `channel2` uses `mode2`, and so forth. Mode strings are either:

- **r** — Configure the channel to read data from the processor.
- **w** — Configure the channel for writing data to the processor.

`open(rx,channel,mode)` opens a new channel to the processor associated with the link `rx`. The new channel uses the `channel` string and is configured for reading or writing according to the `mode` string.

`open(cc,filename,filetype,timeout)` loads `filename` into CCS IDE. `filename` can be the full path to the file or, if the file is in the current CCS IDE working directory, you can use a relative path, such as the name of the file.

Use `cd` to determine or change the CCS IDE working directory. You use the `filetype` option to override the default file extension. The `filetype` string, 'project', is the only string that works in this function syntax.

filetype String	Extension	Description
project	.c, .a*, .s*, .o*, .lib, .cmd, .mak	CCS IDE project files

To determine how long MATLAB software waits for `open` to load the file into CCS IDE, `timeout` sets the upper limit, in seconds, for the period MATLAB software waits for the load. If MATLAB software waits more than `timeout` seconds, `load` returns immediately with a timeout error. Returning a timeout error does not suspend the operation; it stops MATLAB software from waiting for confirmation for the operation completion.

`open(cc,filename,filetype)` loads `filename` into CCS IDE. `filename` can be the full path to the file or, if the file is in the current CCS IDE working directory, you can use a relative path, such as the name of the file. Use the `cd` function to determine or change your CCS IDE working directory. You use the `filetype` option to override the default file extension. Refer to the previous syntax for more information about `filetype`. When you omit the `timeout` option in this syntax, MATLAB software uses the global timeout set in `cc`.

`open(cc,filename)` loads `filename` into CCS IDE. `filename` can be the full path to the file or, if the file is in the current CCS IDE working directory, you can use a relative path, such as the name of the file. Use the `cd` function to determine or change the CCS IDE working directory. You use the `filetype` option to override the default file extension. Refer to the previous syntax for more information about `filetype`. When you omit the `filetype` and `timeout` options in this syntax, MATLAB software uses the global timeout set in `cc`, and derives the

open

file type from the extension in *filename*. Refer to the previous syntax descriptions for more information on the input options.

Note You cannot write to or read from channels that you opened but did not enable.

Examples

For RTDX use, `open` forms part of the function pair you use to open and enable a communications channel between MATLAB software and your processor.

```
cc = ticcs;  
rx = cc.rtdx;  
open(rx, 'ichannel', 'w');  
enable(rx, 'ichannel');
```

When you are working with CCS IDE, `open` adopts a different operational form based on your input arguments for *filename* and the optional arguments *filetype* and *timeout*. In the CCS IDE variant, `open` loads the specified file into CCS IDE. For example, to load the tutorial program used in Getting Started with Automation Interface, use the following syntax

```
cc = ticcs;  
cc.load(tutorial_6xevm.out);
```

See Also

`cd`, `dir`, `load`

Purpose

(For CCS) Code execution and stack usage profile report

Note The **tic** and **raw** profile report options that depend on DSP/BIOS will be removed in a future release. Use **report** for all profiling.

Syntax

```
ps=profile(cc, execution , 'format', timeout)
ps=profile(cc, 'execution', 'format')
profile(cc, 'stack', 'action')
```

Description

ps=profile(cc, **execution** , 'format', timeout) returns execution profile measurements from the generated code. Structure **ps** contains the information in either raw form or filtered and formatted into fields.

To use **profile** to assess how your program executes in real-time, complete the following tasks with a Simulink model:

- 1** Enable real-time execution profiling in the configuration parameters and build your model.
- 2** Select whether to profile by task or subsystem.
- 3** Build your model.
- 4** Download your program to the processor.
- 5** Run the program on the processor.
- 6** Stop the running program.
- 7** Use **profile** at the MATLAB command prompt to access the profiling reports.

If your project uses DSP/BIOS, the profiling system uses CLK and STS objects to profile your project. *STS objects* buffer statistics data accesses by statistics functions in the operating system. The objects are a service provided by the DSP/BIOS real-time kernel. For details about STS

objects and DSP/BIOS, refer to your Texas Instruments documentation that came with CCS IDE.

Note Profiling works with and without enabling DSP/BIOS in your project. To use DSP/BIOS, you must install Target Support Package.

To define how to return the profiling information, set the `format` input argument.

format String	Description
raw	Returns an unformatted list of the timing objects (profiling) information. Returns and formats all time-based objects.
report	Returns the same data as the raw option, formatted into an HTML report. Works only on projects that include DSP/BIOS. If you own Target Support Package software, <code>profile(cc, 'execution', 'report')</code> provides more information about code you generate from Simulink software models.
tic	Returns a formatted list of the STS timing objects information. Filters out some of the information returned with the raw option. To be returned by this format, the object must be time-based. Does not return user-defined objects. Use raw to see user-defined objects.

Entries in the next table explain when you can use **raw**, **report**, and **tic** with your projects—whether the format applies to task or atomic subsystem profiling and whether the format applies with DSP/BIOS.

format String	Profiling by Parameter	DSP/BIOS Project	Non-DSP/BIOS Project
raw	Task	No	No
	Atomic Subsystem	Yes	No
report	Task	No	Yes
	Atomic Subsystem	Yes	Yes
tic	Task	No	No
	Atomic Subsystem	Yes	No

The following examples show the different report formats that **raw**, **report**, and **tic** provide:

- raw

```

    cpuload: 0
    error: 0
    avgperiod: 1000
    rate: 1000
    obj: [4x1 struct]

for k=1:length(ps.obj),disp(k),disp(ps.obj(k)),end;
1

    name: 'KNL_swi'
    units: 'Hi Time'
    max: 1564
    total: 10644
    avg: 367.0345
    pdfactor: 0.0075
    count: 29

```

2

```
    name: 'processing_SWI'  
    units: 'Hi Time'  
    max: 1528  
    total: 3052  
    avg: 1526  
pdfactor: 0.0075  
count: 2
```

3

```
    name: 'TSK_idle'  
    units: 'Hi Time'  
    max: -2.1475e+009  
    total: 0  
    avg: 0  
pdfactor: 0.0075  
count: 0
```

4

```
    name: 'IDL_busyObj '  
    units: 'User Def '  
    max: -2.1475e+009  
    total: 0  
    avg: 0  
pdfactor: 0  
count: 0
```

- report (without DSP/BIOS)
 “Profiling Execution by Tasks” on page 4-10
- report (with DSP/BIOS)
 “Profiling Execution by Subsystems” on page 4-12
- tic

```
    cpuload: 0
        obj: [3x1 struct]

ps.obj(1)

ans =

    name: 'KNL_swi'
    units: 'Hi Time'
        max: 1.1759e-005
        avg: 2.7597e-006
    count: 29

for k=1:length(ps.obj),disp(k),disp(ps.obj(k)),end;
1

    name: 'KNL_swi'
    units: 'Hi Time'
        max: 1.1759e-005
        avg: 2.7597e-006
    count: 29

2

    name: 'processing_SWI'
    units: 'Hi Time'
        max: 1.1489e-005
        avg: 1.1474e-005
    count: 2

3

    name: 'TSK_idle'
    units: 'Hi Time'
        max: -16.1465
        avg: 0
    count: 0
```

When you choose **raw**, returned variable `ps` contains an undocumented list of the information provided by CCS IDE. The **tic** option provides the same information in `ps`, as a collection of fields.

Fields in <code>ps</code>	Description
<code>ps.cpuload</code>	Execution time in percent of total time spent out of the idle task.
<code>ps.obj</code>	Vector of defined STS objects in the project.
<code>ps.obj(n).name</code>	User-defined name for an STS object <code>sts(n)</code> . Value for <code>n</code> ranges from 1 to the number of defined STS objects.
<code>ps.obj(n).units</code>	Either <code>Hi Time</code> or <code>Low Time</code> . Describes the timer applied by this STS object, high- or low- resolution time based.
<code>ps.obj(n).max</code>	Maximum measured profile period for <code>sts(n)</code> , in seconds.
<code>ps.obj(n).avg</code>	Average measured profile period for <code>sts(n)</code> , in seconds.
<code>ps.obj(n).count</code>	Number of STS measurements taken while executing the program.

Note When you enable DSP/BIOS in your project, your CLK and STS must be configured correctly for the profiling information to be accurate. Use the DSP/BIOS configuration file to add and configure CLK and STS objects for your project.

With projects that you generate that use DSP/BIOS, the `report` option creates a report that contains all of the information provided

by the other options, plus additional data that comes from DSP/BIOS instrumentation in the project.

`ps=profile(cc, 'execution', 'format')` defaults to the timeout period specified in the `ticcs` object `cc`.

`profile(cc, 'stack', 'action')` returns the CPU stack usage from your application. `action` defines the stack use profile operation and accepts one of the strings in the following table.

action String	Description
setup	Initializes the CPU stack with known patterns. Writes 0xA5 to the stack addresses on C6000 processors and 0xA5A5 on C2000 and C5000 processors.
report	Returns the report of the stack usage from running your application.

The MATLAB output from profiling the system stack has the elements described in the following table.

Report Entry	Units	Description
System Stack	Minimum Addressable Unit (MAU)	Maximum number of MAUs used and the total MAUs allocated for the stack.
name	String for the stack name	Lists the name assigned to the stack.
startAddress	Decimal address and page	Lists the address of the stack start and the memory page.
endAddress	Decimal address and page	Lists the address of the end of the stack and the memory page.

Report Entry	Units	Description
stackSize	Addresses	Reports number of address locations, in MAUs, allocated for the stack.
growthDirection	Not applicable	Reports whether the stack grows from the lower address to the higher address (ascending) or from higher to lower (descending).

To use `profile` to assess how your program uses the stack, complete the following tasks with a Simulink model or manually written code:

- 1** Build your model with real-time execution profiling enabled in the configuration parameters. Skip this step for custom code.
- 2** Download your program to the processor.
- 3** Run the program on the processor.
- 4** Stop the running program.
- 5** Use `profile` at the MATLAB command prompt to access the profiling reports.

For more information about using stack profiling, refer to “System Stack Profiling” on page 4-17.

Using Profiling

The following items affect your ability to profile project execution and stack usage:

Execution profiling works on code you generate from a Simulink model. You cannot profile manually written code that you provide in your project.

Stack profiling works with both model-generated code and your custom code.

Stack profiling does not work when your project uses DSP/BIOS. You get an error when you profile the system stack with DSP/BIOS enabled.

To use DSP/BIOS, you must install Target Support Package software.

For more information about enabling and using execution profiling, refer to “Profiling Code Execution in Real-Time” on page 4-9.

Examples

This example presents two forms of the data returned by `profile`—`tic` and `raw`. The generated code did not include DSP/BIOS.

Running `profile` returns structure `ps` containing profiling data gathered while your program ran. Stop the running program before you request the profile data.

```
ps=profile(cc,'execution','tic')

ps =

    cpload: 0
         obj: [3x1 struct]

ps.obj(1)

ans =

    name: 'KNL_swi'
  units: 'Hi Time'
    max: 1.1759e-005
    avg: 2.7597e-006
   count: 29
```

profile

```
for k=1:length(ps.obj),disp(k),disp(ps.obj(k)),end;
1

    name: 'KNL_swi'
units: 'Hi Time'
    max: 1.1759e-005
    avg: 2.7597e-006
count: 29

2

    name: 'processing_SWI'
units: 'Hi Time'
    max: 1.1489e-005
    avg: 1.1474e-005
count: 2

3

    name: 'TSK_idle'
units: 'Hi Time'
    max: -16.1465
    avg: 0
count: 0
```

Omitting the `format` option caused `profile` to return the data fully formatted and slightly filtered. Adding the `raw` option to `profile` returns the same information without filtering any of the returned data.

```
ps=profile(cc,'execution','raw')

ps =

    cpuload: 0
    error: 0
    avgperiod: 1000
    rate: 1000
```

```
obj: [4x1 struct]
for k=1:length(ps.obj),disp(k),disp(ps.obj(k)),end;
1
    name: 'KNL_swi'
  units: 'Hi Time'
    max: 1564
  total: 10644
    avg: 367.0345
pdfactor: 0.0075
  count: 29

2
    name: 'processing_SWI'
  units: 'Hi Time'
    max: 1528
  total: 3052
    avg: 1526
pdfactor: 0.0075
  count: 2

3
    name: 'TSK_idle'
  units: 'Hi Time'
    max: -2.1475e+009
  total: 0
    avg: 0
pdfactor: 0.0075
  count: 0

4
    name: 'IDL_busyObj'
  units: 'User Def'
```

```
max: -2.1475e+009
total: 0
avg: 0
pdfactor: 0
count: 0
```

Your results can differ from this example depending on your computer and processor. The raw-format data in this example includes one extra timing object—IDL_busyObj. As defined in the .tcf file, this object is not time based (**Units** is 'User Def'). Specifying **tic** does not return the IDL_busyObj object.

The following example demonstrates setting up and profiling the system stack. The ticcs object cc must exist in your MATLAB workspace and your application must be loaded on your processor. This example comes from a C6713 simulator.

```
profile(cc,'stack','setup') % Set up processor stack--write 0xA5 to stack addresses.
```

```
Maximum stack usage:
```

```
System Stack: 0/1024 (0%) MAUs used.
```

```
name: System Stack
startAddress: [512 0]
endAddress: [1535 0]
stackSize: 1024 MAUs
growthDirection: ascending
```

```
run(cc)
```

```
halt(cc)
```

```
profile(cc,'stack','report') % Request stack use report.
```

```
Maximum stack usage:
```

```
System Stack: 356/1024 (34.77%) MAUs used.
```

```
name: System Stack
startAddress: [512 0]
endAddress: [1535 0]
stackSize: 1024 MAUs
growthDirection: ascending
```

See Also [ticcs](#)

read

Purpose (For CCS) Data from memory on processor or in CCS

Syntax

```
mem = read(cc,address,'datatype',count,timeout)
mem = read(cc,address,'datatype',count)
mem = read(cc,address,'datatype')
```

Description **ticcs Object Syntaxes**

`mem = read(cc,address,'datatype',count,timeout)` returns data from the processor referred to by `cc`. The `address`, `count`, and `datatype` input arguments define the memory block to be read. The data block to read begins at the memory location defined by `address`. `count` determines the number of values to read, starting at `address`. `datatype` defines the format of the raw data stored in the referenced memory block.

To check values in memory on a running processor, such as values that change during processing, insert one or more breakpoints in the project code and perform the read operation while the processor code is paused at one of the breakpoints. After you read the data, release the breakpoint.

Note

Do not attempt to read data from the processor while it is running. Reading data from a running process can produce incorrect values.

`read` uses the `datatype` parameter to determine the number of bytes to read per stored value. `timeout` is an optional input argument you use to specify when to terminate long read processes and data transfers. For details about each input parameter, read the following descriptions.

address — `read` uses `address` to define the beginning of the memory block to read. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector [*location*, *page*], a string, or a

decimal value. When the processor has only one memory page, as is true for many digital signal processors, the value of the page portion of the memory address is 0. By default, `ticcs` sets the page to 0 at creation if you omit the page property as an input argument.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
1F	String	Offset is 31 decimal on the page referred to by <code>cc.page</code>
10	Decimal	Offset is 10 decimal on the page referred to by <code>cc.page</code>
[18,1]	Vector	Offset is 18 decimal on memory page 1 (<code>cc.page = 1</code>)

To specify the address in hexadecimal format, enter the address property value as a string. `read` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. Note that when you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc.page`.

`count` — a numeric scalar or vector that defines the number of `datatype` values to read. Entering a scalar for `count` causes `read` to return `mem` as a column vector which has `count` elements. `count` can be a vector to define multidimensional data blocks. The elements of `count` define the dimensions of the data matrix returned in `mem`. The following table shows examples of input arguments to `count` and how `read` responds.

read

Input	Response
<code>n</code>	Read <code>n</code> values into a column vector. Return the vector in mem.
<code>[m,n]</code>	Read (<code>m*n</code>) values from memory into an <code>m</code> -by- <code>n</code> matrix in column major order. Return the matrix in mem.
<code>[m,n,p,...]</code>	Read (<code>m*n*p*...</code>) values from the processor memory in column major order. Return the data in an <code>m</code> -by- <code>n</code> -by- <code>p</code> -by... multidimensional matrix and return the matrix in mem.

`datatype` — a string that represents a MATLAB data type. The total memory block size is derived from the value of `count` and the specified `datatype`. `datatype` determines how many bytes to check for each memory value. `read` supports the following data types:

<code>datatype</code> String	Description
'double'	Double-precision floating point values
'int8'	Signed 8-bit integers
'int16'	Signed 16-bit integers
'int32'	Signed 32-bit integers
'single'	Single-precision floating point data
'uint8'	Unsigned 8-bit integers
'uint16'	Unsigned 16-bit integers
'uint32'	Unsigned 32-bit integers

To limit the time that `read` spends transferring data from the processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the read operation. You might find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, `read` defaults to the global timeout defined in `cc`.

Working with Negative Values

Writing a negative value causes the data written to be saturated because `char` is unsigned on the processor. Hence, a 0 (a NULL) is written instead. A warning results as well, as this example shows.

```
cc = ticcs;
ff = createobj(cc,'g_char'); % Where g_char is in the code.
write(ff,-100);
Warning: Underflow: Saturation was required to fit the data into
an addressable unit.
```

When you try to read the data you wrote, the character being read is 0 (NULL) — so there seems to be nothing returned by the `read` function.

You can demonstrate this by the following code, after writing a negative value as shown in the previous example.

```
readnumeric(x)
ans =
0
read(x) % Reads the NULL character
ans = % Apparently nothing is returned.

double(read(x)) % Read the numeric equivalent of NULL.
ans = % Again, appears not to return a value.
```

`mem = read(cc, address, 'datatype', count)` reads data from memory on the processor referred to by `cc` and defined by the `address`, and `datatype` input arguments. The data block being read begins at the memory location defined by `address`. `count` determines the number of values to be read. When you omit the `timeout` option, `timeout` defaults to the value specified by the `timeout` property in `cc`.

`mem = read(cc, address, 'datatype')` reads the memory location defined by the `address` input argument from the processor memory referred to by `cc`. The data block being read begins at the memory location defined by `address`. When you omit the `count` option, `count`

read

defaults to a value of 1. This syntax reads one memory location of datatype.

Note To ensure seamless read operation, use `address` to extract address values that are compatible with the alignment required by your processor. `read` does not force data type alignment in your processor memory.

Certain combinations of `address` and `datatype` are difficult for some processors to use. To ensure seamless read operation, use the `address` function to extract address values that are compatible with the alignment required by your processor.

Like the `isreadable`, `iswritable`, and `write` functions, `read` checks for valid address values. Illegal address values are any address space larger than the available space for the processor — 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When `read` identifies an illegal address, it returns an error message stating that the address values are out of range.

Examples

`read` reads data that you wrote to the processor.

```
cc = ticcs;  
indata = 1:25;  
write(cc,0,indata,30);  
outdata=read(cc,0,'double',25,10)
```

```
outdata =
```

```
Columns 1 through 13
```

```
1    2    3    4    5    6    7    8    9   10   11   12   13
```

```
Columns 14 through 25
```

```
14 15 16 17 18 19 20 21 22 23 24 25
```

outdata now contains the values in indata, returned from the processor.

As a further demonstration of read, try the following functions after you create a link cc and load an appropriate program to your processor. To perform the first example, var must exist in the symbol table loaded in CCS.

- Read one 16-bit integer at the location of processor symbol var.

```
mlvar = read(cc,address(cc,'var'),'int16')
```

- Read 100 32-bit integers from address f000 (hexadecimal) and plot the data.

```
mlplt = read(cc,'f000','int32',100)
plot(double(mlplt))
```

- Increment the integer value stored at address 10 (decimal) of the processor.

```
cc = ticcs;
ainc = 10
mlinc = read(cc,ainc,'int32')
mlinc = int32(double(mlinc)+1)
cc.write(ainc,mlinc)
```

See Also

isreadable, symbol, write

readmat

Purpose

(For CCS) Matrix of data from RTDX channel

Note Support for readmat on C5000 and C6000 processors will be removed in a future version.

Syntax

```
data = readmat(rx,channelname,'datatype',siz,timeout)
data = readmat(rx,channelname,'datatype',siz)
```

Description

`data = readmat(rx,channelname,'datatype',siz,timeout)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

Before you read from a channel, open and enable the channel for read access.

Replace `channelname` with the string you specified when you opened the desired channel. `channelname` must identify a channel that you defined in the program loaded on the processor.

You cannot read data from a channel you have not opened and configured for read access. If necessary, use the RTDX tools provided in CCS IDE to determine which channels exist for the loaded program.

`data` contains a matrix whose dimensions are given by the input argument vector `siz`, where `siz` can be a vector of two or more elements. To operate properly, the number of elements in the output matrix `data` must be an integral number of channel messages.

When you omit the `timeout` input argument, `readmat` reads messages from the specified channel until the output matrix is full or the global `timeout` period specified in `rx` elapses.

Caution If the timeout period expires before the output data matrix is fully populated, you lose all the messages read from the channel to that point.

MATLAB software supports reading five data types with `readmat`:

datatype String	Data Format
'double'	Double-precision floating point values. 64 bits.
'int16'	16-bit signed integers
'int32'	32-bit signed integers
'single'	Single-precision floating point values. 32 bits.
'uint8'	Unsigned 8-bit integers

`data = readmat(rx,channelname,'datatype',siz)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

Examples

In this data read and write example, you write data to the processor through the CCS IDE. You can then read the data back in two ways — either through `read` or through `readmsg`.

To duplicate this example you need to have a program loaded on the processor. The channels listed in this example, `ichannel` and `ochannel`, must be defined in the loaded program. If the current program on the processor defines different channels, replace the listed channels with your current ones.

```
cc = ticcs;
rx = cc.rtdx;
open(rx,'ichannel','w');
enable(rx,'ichannel');
```

readmat

```
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
write(cc,0,indata,30);
outdata=read(cc,0,'double',25,10)
```

```
outdata =
```

```
Columns 1 through 13
```

```
1  2  3  4  5  6  7  8  9 10 11 12 13
```

```
Columns 14 through 25
```

```
14 15 16 17 18 19 20 21 22 23 24 25
```

Now use RTDX to read the data into a 5-by-5 array called `out_array`.

```
out_array = readmat('ochannel','double',[5 5])
```

See Also

`readmsg`, `writemsg`

Purpose (For CCS) Read messages from specified RTDX channel

Note Support for readmsg on C5000 and C6000 processors will be removed in a future version.

Syntax

```
data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)
data = readmsg(rx,channelname,'datatype',siz,nummsgs)
data = readmsg(rx,channelname,datatype,siz)
data = readmsg(rx,channelname,datatype,nummsgs)
data = readmsg(rx,channelname,datatype)
```

Description

`data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read all the messages present in the queue when you call the `readmsg` function.

Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. For example, when `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `m`-by-`n` matrices in `data`. Each output matrix in `data` must have the same number of elements (`m` x `n`) as the number of elements in each message.

You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports strings that define the type of data you are expecting, as shown in the following table.

datatype String	Specified Data Type
'double'	Floating point data, 64-bits (double-precision).
'int16'	Signed 16-bit integer data.
'int32'	Signed 32-bit integers.

readmsg

datatype String	Specified Data Type
'single'	Floating-point data, 32-bits (single-precision).
'uint8'	Unsigned 8-bit integers.

When you include the `timeout` input argument in the function, `readmsg` reads messages from the specified queue until it receives `nummsgs`, or until the period defined by `timeout` expires while `readmsg` waits for more messages to be available.

When the desired number of messages is not available in the queue, `readmsg` enters a wait loop and stays there until more messages become available or `timeout` seconds elapse. The `timeout` argument overrides the global timeout specified when you create `rx`.

`data = readmsg(rx,channelname,'datatype',siz,nummsgs)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read all the messages present in the queue when you call the `readmsg` function.

Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. When `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `n`-by-`m` matrices in `data`.

Each output matrix in `data` must have the same number of elements (`m` x `n`) as the number of elements in each message.

You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports six strings that define the type of data you are expecting.

`data = readmsg(rx,channelname,datatype,siz)` reads one data message because `nummsgs` defaults to one when you omit the input argument. `readmsg` returns the message as a row vector in `data`.

`data = readmsg(rx,channelname,datatype,nummsgs)` reads the number of messages defined by `nummsgs`. `data` becomes a cell array of row matrices, `data = {msg1,msg2,...,msg(nummsgs)}`, because `siz` defaults to `[1,nummsgs]`; each returned message becomes one row matrix in the cell array.

Each row matrix contains one element for each data value in the current message `msg# = [element(1), element(2),...,element(l)]` where `l` is the number of data elements in message. In this syntax, the read messages can have different lengths, unlike the previous syntax options.

`data = readmsg(rx,channelname,datatype)` reads one data message, returning a row vector in `data`. All of the optional input arguments—`nummsgs`, `siz`, and `timeout`—use their default values.

In all calling syntaxes for `readmsg`, you can set `siz` and `nummsgs` to empty matrices, causing them to use their default values—`nummsgs = 1` and `siz = [1,1]`, where `l` is the number of data elements in the read message.

Caution If the timeout period expires before the output data matrix is fully populated, you lose all the messages read from the channel to that point.

Examples

```
cc = ticcs;
rx = cc.rtdx;
open(rx,'ichannel','w');
enable(rx,'ichannel');
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
write(cc,0,indata,30);
outdata=read(cc,0,'double',25,10)

outdata =
```

readmsg

Columns 1 through 13

```
1  2  3  4  5  6  7  8  9 10 11 12 13
```

Columns 14 through 25

```
14 15 16 17 18 19 20 21 22 23 24 25
```

Now use `RTDX` to read the messages into a 4-by-5 array called `out_array`.

```
number_msgs = msgcount(rx,'ochannel') % Check number of msgs
                                     % in read queue.
out_array = cc.rtdx.readmsg('ochannel','double',[4 5])
```

See Also

`read`, `readmat`, `writemsg`

Purpose (For CCS) Value from processor register

Syntax

```
reg = regread(cc,'regname','represent',timeout)
reg = regread(cc,'regname','represent')
reg = regread(cc,'regname')
```

Description `reg = regread(cc,'regname','represent',timeout)` reads the data value in the `regname` register of the processor and returns the value in `reg` as a double-precision value. For convenience, `regread` converts each return value to the MATLAB software double datatype. Making this conversion lets you manipulate the data in MATLAB software. String `regname` specifies the name of the source register on the processor. `ticcs` object `cc` defines the processor to read from. Valid entries for `regname` depend on your processor. Register names are not case-sensitive — `a0` is the same as `A0`.

For example, the TMS320C6xxx processor family provides the following register names that are valid entries for `regname`:

Register Names	Register Contents
A0, A1, A2,..., A15	General purpose A registers
B0, B1, B2,..., B15	General purpose B registers
PC, ISTEP, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1,..., B15:B14	64-bit general purpose register pairs

Other processors provide other register sets. Refer to the documentation for your processor to determine the registers for the processor.

Note Use `read` (called a *direct memory read*) to read memory-mapped registers.

regread

The `represent` input argument defines the format of the data stored in `regname`. Input argument `represent` takes one of three input strings:

represent String	Description
'2scomp'	Source register contains a signed integer value in two's complement format. This is the default setting when you omit the <code>represent</code> argument.
'binary'	Source register contains an unsigned binary integer.
'ieee'	Source register contains a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are reading from 32 and 64 bit registers on the processor.

To limit the time that `regread` spends transferring data from the processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the read operation. You might find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, `regread` defaults to the global timeout defined in `cc`.

`reg = regread(cc, 'regname', 'represent')` does not set the global timeout value. The timeout value in `cc` applies.

`reg = regread(cc, 'regname')` does not define the format of the data in `regname`.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations at any time during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned

later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables during intermediate times in program operation may not get reflected in the register.

This is true for local variables as well.

One way to see this is to write a line of code that uses the variable and see if the result is consistent.

```
register int a = 100;
int b;
...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or Embedded IDE Link software.

Examples

For the C5xxx processor family, most registers are memory-mapped and consequently are available using `read` and `write`. However, use `regread` to read the PC register. The following command demonstrates how to read the PC register. To identify the processor, `cc` is a link for CCS IDE.

```
cc.regread('PC', 'binary')
```

To tell MATLAB software what datatype you are reading, the string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

In response, MATLAB software displays

```
ans =
    33824
```

regread

For processors in the C6xxx family, `regread` lets you access processor registers directly. To read the value in general purpose register A0, type the following function.

```
treg = cc.regread('A0', '2scomp');
```

`treg` now contains the two's complement representation of the value in A0.

Now read the value stored in register B2 as an unsigned binary integer, by typing

```
cc.regread('B2', 'binary');
```

See Also

`read`, `regwrite`, `write`

Purpose (For CCS) Write data values to registers on processor

Syntax

```
regwrite(cc, 'regname', value, 'represent', timeout)
regwrite(cc, 'regname', value, 'represent')
regwrite(cc, 'regname', value,)
```

Description `regwrite(cc, 'regname', value, 'represent', timeout)` writes the data in `value` to the `regname` register of the processor. `regwrite` converts `value` from its representation in the MATLAB workspace to the representation specified by `represent`. The `represent` input argument defines the format of the data when it is stored in `regname`. Input argument `represent` takes one of three input strings:

represent String	Description
'2scomp'	Write value to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the <code>represent</code> argument.
'binary'	Write value to the destination register as an unsigned binary integer.
'ieee'	Write value to the destination registers as a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are writing to 32- and 64-bit registers on the processor.

Note Use `write` (called a *direct memory write*) to write memory-mapped registers.

String `regname` specifies the name of the destination register on the processor. `cc` defines the processor to write `value` to. Valid entries for `regname` depend on your processor. Register names are not case-sensitive — `a0` is the same as `A0`. For example, the `C6xxx`

regwrite

processor family provides the following register names that are valid entries for `regname`:

Register Names	Register Contents
A0, A1, A2,..., A15	General purpose A registers
B0, B1, B2,..., B15	General purpose B registers
PC, ISTR, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1,..., B15:B14	64-bit general purpose register pairs

Other processors provide other register sets. Refer to the documentation for your processor to determine the registers for the processor.

To limit the time that `regwrite` spends transferring data to the processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You might find this useful for limiting prolonged data transfer operations.

If you omit the `timeout` input argument in the syntax, `regwrite` defaults to the global timeout defined in `cc`. If the write operation exceeds the time specified, `regwrite` returns with a timeout error. Generally, timeout errors do not stop the register write process. The write process stops while waiting for CCS IDE to respond that the write operation is complete.

`regwrite(cc, 'regname', value, 'represent')` omits the `timeout` input argument and does not change the `timeout` value specified in `cc`.

`regwrite(cc, 'regname', value,)` omits the `represent` input argument. Writing the data does not reformat the data written to `regname`.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations at any time during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables during intermediate times in program operation may not get reflected in the register.

This is true for any local variables as well.

One way to see this is to write a line of code that uses the variable and see if result is consistent.

```
register int a = 100;
int b;
...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or Embedded IDE Link software.

Examples

To write a new value to the PC register on a C5xxx family processor, enter

```
regwrite(cc, 'pc', hex2dec('100'), 'binary')
```

specifying that you are writing the value 256 (the decimal value of 0x100) to register `pc` as binary data.

To write a 64-bit value to a register pair, such as B1:B0, the following syntax specifies the value as a string representation, and processor registers.

regwrite

```
regwrite(cc, 'b1:b0', hex2dec('1010'), 'ieee')
```

Registers B1:B0 now contain the value 4112 in double-precision format.

See Also

read, regread, write

Purpose (For CCS) Reload most recent program file to processor signal processor

Syntax
`s = reload(cc,timeout)`
`s = reload(cc)`

Description `s = reload(cc,timeout)` resends the most recently loaded program file to the processor. If you have not loaded a program file in the current session (so there is no previously loaded file), `reload` returns the null entry `[]` in `s` indicating that it could not load a file to the processor. Otherwise, `s` contains the full path name to the program file. After you reset your processor or after any event produces changes in your processor memory, use `reload` to restore the program file to the processor for execution.

To limit the time CCS IDE spends trying to reload the program file to the processor, `timeout` specifies how long the load process can take. If the load process exceeds the timeout limit, CCS IDE stops trying to load the program file and returns an error stating that the time period expired. Exceeding the allotted time for the reload operation usually indicates that the reload was successful but CCS IDE did not receive confirmation before the timeout period passed.

`s = reload(cc)` reloads the most recent program file, using the `timeout` value set when you created link `cc`, the global timeout setting.

Using reload with Multiprocessor Boards

When your board contains more than one processor, `reload` calls the reloading function for each processor represented by `cc`, reloading the most recently loaded program on each processor.

This is the same as calling `reload` for each processor individually through `ticcs` objects for each one.

Examples After you create an object that connects to CCS, use the available methods to reload your most recently loaded project. If you have not loaded a project in this session, `reload` returns an error and an empty value for `s`. Loading a project eliminates the error.

reload

```
cc=ticcs;
s=reload(cc,23)
Warning: No action taken - load a valid Program file before
you reload...

s =

    ''

open(cc,'D:\ti\tutorial\sim62xx\gelsolid\hellodsp.pjt',...
'project')

build(cc)

load(cc,'hellodsp.pjt')
halt(cc)
s=reload(cc,23)

s =

D:\ti\tutorial\sim62xx\gelsolid\Debug\hellodsp.out
```

See Also

cd, load, open

Purpose	(For CCS) Remove file from active CCS IDE project
Syntax	<pre>remove(cc, 'filename') remove(cc, 'gelfilename')</pre>
Description	<p><code>remove(cc, 'filename')</code> deletes the file specified by <code>filename</code> from the active project in CCS IDE. You can remove files that exist in the active project only. <code>filename</code> must match the name of an existing file exactly to remove the file.</p> <p><code>remove(cc, 'gelfilename')</code> deletes the file specified by <code>gelfilename</code> from the active project in CCS IDE. You can remove files that exist in the active project only. <code>gelfilename</code> must match the name of an existing file exactly to remove the file.</p>
Examples	<p>After you have a project in CCS IDE, you can delete files from it using <code>remove</code> from the MATLAB software command line. For example, build a project and load the resulting <code>.out</code> file. With the project build complete, load your <code>.out</code> file by typing</p> <pre>load(cc, 'filename.out')</pre> <p>Now remove one file from your project, such as the GEL file.</p> <pre>remove(cc, 'gelfilename')</pre> <p>You see in CCS IDE that the GEL file no longer appears in the GEL files folder in CCS.</p>
See Also	<code>activate</code> , <code>add</code> , <code>cd</code> , <code>open</code>

reset

Purpose (For CCS) Reset processor

Syntax `reset(cc, timeout)`
`reset(cc)`

Description `reset(cc, timeout)` stops program execution on the processor and asynchronously performs a processor reset, returning all processor register contents to their power up settings. The `reset` function returns after the processor halts.

To allow you to determine how long `reset` waits for the processor to halt, input option `timeout` lets you set the waiting period in seconds. After you use `reset`, the routine returns after the processor halts or after `timeout` seconds elapses, whichever comes first.

`reset(cc)` stops program execution on the processor and asynchronously performs a processor reset, returning all processor register contents to their power up settings. The `reset` function returns after the processor halts. `reset` uses the global timeout value defined in `cc` to determine how long to wait for the processor to halt before returning. Use `get` to examine the global timeout value for the link.

Use `run` to restart the program loaded on the processor.

Compare to `halt` which does not reset the processor after the program stops running.

Using reset with Multiprocessor Boards

When your board contains more than one processor, `reset` calls the processor resetting function for each processor represented by `cc`, resetting each processor.

This is the same as calling `reset` for each processor individually through `ticcs` objects for each one.

Note that the `run` and `halt` methods still apply as mentioned earlier in this section.

See Also `halt`, `restart`, `run`

Purpose	(For CCS) Restore program counter to entry point for current program
Syntax	<code>restart(cc,timeout)</code> <code>restart(cc)</code>
Description	<p><code>restart(cc,timeout)</code> halts the processor immediately and resets the program counter (PC) to the program entry point for the loaded program. Use <code>run</code> to execute the program after you use <code>restart</code>. <code>restart</code> does not execute the program after resetting the PC. <code>timeout</code> allows you to specify how long <code>restart</code> waits for the processor to stop and return the PC to the program entry point. Specify the value for <code>timeout</code> in seconds. After you use <code>restart</code>, the restart routine returns after resetting the PC or after <code>timeout</code> seconds elapse, whichever comes first. If the timeout period expires, <code>restart</code> returns a timeout error.</p> <p><code>restart(cc)</code> halts the processor immediately and resets the PC to the program entry point for the loaded program. Use <code>run</code> to execute the program after you use <code>restart</code>. <code>restart</code> does not execute the program after resetting the PC. When you omit the <code>timeout</code> argument, <code>restart</code> uses the global default timeout period defined in <code>cc</code> to determine how long to wait for the processor to stop and the PC to be reset to the program entry point.</p> <p>Using restart with Multiprocessor Boards</p> <p>When your board contains more than one processor, <code>restart</code> calls the processor restarting function for each processor represented by <code>cc</code>, restarting the program loaded on each processor.</p> <p>This is the same as calling <code>restart</code> for each processor individually through <code>ticcs</code> objects for each one.</p>
Examples	<p>When you are developing algorithms for your processor, <code>restart</code> becomes a particularly useful function. Rather than resetting the processor after each algorithm test, use the <code>restart</code> function to return the program counter to the program entry point. Because <code>restart</code> restores your local variables to their initial settings, but does not reset the processor, you are ready to rerun your algorithm with new values.</p>

restart

When your process gets lost or halts, `restart` is a quick way to restore your program.

See Also

`halt`, `isrunning`, `run`

Purpose (For CCS) Execute program loaded on processor

Syntax

```
run(cc, 'state', timeout)
run(cc, 'main')
run(cc, 'tofunc', 'functionname')
```

Description `run(cc, 'state', timeout)` starts to execute the program loaded on the processor referred to by `cc`. Program execution starts from the location of the program counter. After starting program execution, the input argument `state` determines when you regain program control.

To define the action of `run`, `state` accepts strings that set the state of the processor:

state String	Run Action
'main'	Reset the program counter then run the program until the PC reaches <code>main</code> . Stop at <code>main</code> .
'run'	Start to execute the program. Wait until the program is running, then return. The program continues to run. If you omit the <code>state</code> argument, <code>run</code> defaults to this setting. Sets the processor to the running state and returns. This is useful when you want to continue to work in MATLAB software while the processor executes a program.
'runtohalt'	Start to execute the program. Wait to return until the program encounters a breakpoint or the program execution terminates. Sets the processor to the running state and returns when the processor halts.

state String	Run Action
'tofunc'	Run the program from the current position of the program counter to the start of a specified function <code>functionname</code> .
'tohalt'	Changes the state of a running process to <code>runtohalt</code> , and waits for the processor to halt before returning. Use this when you want to stop a running process cleanly. If the processor is already stopped when you use this state setting, <code>run</code> returns immediately.

The `timeout` input argument specifies how long MATLAB software waits for the connection to the processor or the response to a command to return completed.

After you use `run`, the routine returns after confirming that the program started to execute, or after `timeout` seconds elapses, whichever comes first. If the timeout period expires, `run` returns a timeout error.

`run(cc, 'main')` resets the program counter in your project then runs the program linked to `cc` until the counter reaches the start of `main`.

`run(cc, 'tofunc', 'functionname')` runs the program from the current position of the program counter until the counter reaches the function `functionname`. Compare this to `run(cc, 'main')` which resets the program counter before executing the program. Using the `tofunc` option does not reset the program counter.

Using run with Multiprocessor Boards

When your board contains more than one processor, `run` calls the program running function for each processor represented by `cc`, running the program loaded on each processor.

This is the same as calling `run` for each processor individually through `ticcs` objects for each one. The other information about `run` on a single processor applies to each processor in the multiple processor cases.

Examples

After you build and load a program to your processor, use `run` to start execution.

```
cc = ticcs('boardnum',0,'procnum',0); % Create a link to CCS
                                     % IDE.
cc.load('tutorial_6xevm.out'); % Load an executable file to the
                               % processor.
cc.rtdx.configure(1024,4); % Configure four buffers for data
                           % transfer needs.

cc.rtdx.open('ichan','w'); % Open RTDX channels for read and
                           % write.

cc.rtdx.enable('ichan');
cc.rtdx.open('ochan','r');
cc.rtdx.enable('ochan');

cc.restart; % Return the PC to the beginning of the current
            % program.

cc.run('run'); % Run the program to completion.
```

This example uses a tutorial program included with Embedded IDE Link. Set your CCS IDE working directory to be the one that holds your project files. The `load` function uses the current working directory unless you provide a full path name in the input arguments.

Rather than using the dot notation to access the RTDX functions, you can create an alias to the `cc` link and use the alias in later commands. Thus, if you add the line

```
rx = cc.rtdx;
```

to the program, you can replace

```
cc.rtdx.configure(1024,4);
```

with

run

```
configure(rx,1024,4);
```

See Also

halt, isrunning, restart

Purpose (For CCS) Save files and projects in CCS IDE

Note `save(cc,filename,'text')` produces an error.

Syntax `save(cc,'filename','type')`

Description `save(cc,'filename','type')` save the file in CCS IDE identified by `filename` of type `'type'`. `type` identifies the type of file to save, either project files when you use `'project'` for type, or text files when you use `'text'` for the type option. To save a specific file in CCS IDE, `filename` must match the name of the file to save exactly. If you replace `filename` with `'all'`, `save` writes every open file whose type matches the type option. File types recognized by `save` include these extensions.

type String	Affected files
'project'	Project files with the .pjt extension.
'text'	All files with these extensions — a*, .c, .cc, .ccx, .tcf, .cmd, .cpp, .lib, .o*, .rcp, and .s*. Note that 'text' does not save .cfg files.

When you replace `filename` with the null entry `[]`, `save` writes to storage the current active file window in CCS IDE, or the active project when you specify `project` for the `type` option.

Examples To clarify the different `save` options, here are commands that save open files or projects in CCS IDE.

Command	Result
<code>save(cc,'all','project')</code>	Save all open projects in CCS IDE.
<code>save(cc,'my.pjt','project')</code>	Save the project <code>my.pjt</code> .
<code>save(cc,[],project')</code>	Save the active project.

save

Command	Result
<code>save(cc, 'all', 'text')</code>	Save all open text files. Includes source files, libraries, command files, and others.
<code>save(cc, 'my_source.cpp', 'text')</code>	Save the text file <code>my_source.cpp</code> .
<code>save(cc, [], 'text')</code>	Save the active file window.

See Also

`add`, `cd`, `close`, `open`

Purpose (For CCS) Program symbol table from CCS IDE

Syntax `s = symbol(cc)`

Description `s = symbol(cc)` returns the symbol table for the program loaded in CCS IDE. `symbol` only applies after you load a processor program file. `s` is an array of structures where each row in `s` presents the symbol name and address in the table. Therefore, `s` has two columns; one is the symbol name, and the other is the symbol address and symbol page. For example, this table shows a few possible elements of `s`, and their interpretation.

s Structure Field	Contents of the Specified Field
<code>s(1).name</code>	String reflecting the symbol entry name.
<code>s(1).address(1)</code>	Address or value of symbol entry.
<code>s(1).address(2)</code>	Memory page for the symbol entry. For TI C6xxx processors, the page is 0.

You can use field `address` in `s` as the `address` input argument to `read` and `write`.

If you use `symbol` and the symbol table does not exist, `s` returns empty and you get a warning message.

Symbol tables are a portion of a COFF object file that contains information about the symbols that are defined and used by the file. When you load a program to the processor, the symbol table resides in CCS IDE. While CCS IDE may contain more than one symbol table at a time, `symbol` accesses the symbol table belonging to the program you last loaded on the processor.

Examples Demonstrating this function requires that you load a program file to your processor. In this example, build and load Embedded IDE Link demo program `c6711dskafxr`. Start by entering `c6711dskafxr` at the MATLAB software prompt.

symbol

```
c6711dskafxr;
```

Now set the simulation parameters for the model and build the model to your processor. With the model loaded on your processor, use `symbol` to return the entries stored in the symbol table in CCS IDE.

```
cc = ticcs;  
s = symbol(cc);
```

`s` contains all the symbols and their addresses, in a structure you can display with the following code:

```
for k=1:length(s),disp(k),disp(s(k)),end;
```

MATLAB software lists the symbols from the symbol table in a column.

See Also

`load`, `run`

Purpose (For CCS) Create object that refers to CCS IDE

Syntax
`cc = ticcs`
`cc = ticcs('propertyname','propertyvalue',...)`

Description `cc = ticcs` returns a `ticcs` object in `cc` that MATLAB software uses to communicate with the default processor. In the case of no input arguments, `ticcs` constructs the object with default values for all properties. CCS IDE handles the communications between MATLAB software and the selected CPU. When you use the function, `ticcs` starts CCS IDE if it is not running. If `ticcs` opened an instance of the CCS IDE when you issued the `ticcs` function, CCS IDE becomes invisible after Embedded IDE Link creates the new object.

Note When `ticcs` creates the object `cc`, it sets the working directory for CCS IDE to be the same as your MATLAB software working directory. When you create files or projects in CCS IDE, or save files and projects, this working directory affects where you store the files and projects.

Each object that accesses CCS IDE comprises two objects—a `ticcs` object and an `rt dx` object—that include the following properties.

Object	Property Name	Property	Default	Description
ticcs	'apiversion'	API version	N/A	Defines the API version used to create the link
	'proctype'	Processor Type	N/A	Specifies the kind of processor on the board
	'procname'	Processor Name	CPU	Name given to the processor on the board to which this object links

Object	Property Name	Property	Default	Description
	'status'	Running	No	Status of the program currently loaded on the processor
	'boardnum'	Board Number	0	Number that CCS assigns to the board. Used to identify the board
	'procnum'	Processor number	0	Number the CCS assigns to a processor on a board
	'timeout'	Default timeout	10.0 s	Specifies how long MATLAB software waits for a response from CCS after issuing a request. This also applies when you try to construct a ticcs object. The create process waits for this timeout period for the connection to the processor to complete. If the timeout period expires, you get an error message that the connection to the processor failed and MATLAB software could not create the ticcs object.

Object	Property Name	Property	Default	Description
rt dx	'timeout'	Timeout	10.0 s	Specifies how long CCS waits for a response from the processor after requesting data
	'numchannels'	Number of open channels	0	The number of open channels using this link
type	type	Defined types in the object	Void, Float, Double, Long, Int, Short, Char	List of the C data types in the project cc accesses. Use add to include your C type definitions to the list

`cc = ticcs('propertyname', 'propertyvalue', ...)` returns a handle in `cc` that MATLAB software uses to communicate with the specified processor. CCS handles the communications between the MATLAB environment and the CPU.

MATLAB software treats input parameters to `ticcs` as property definitions. Each property definition consists of a property name/property value pair.

Two properties of the `ticcs` object are read only after you create the object:

- 'boardnum' — the identifier for the installed board selected from the active boards recognized by CCS. If you have one board, use the default property value 0 to access the board.
- 'procnum' — the identifier for the processor on the board defined by `boardnum`. On boards with more than one processor, use this value to specify the processor on the board. On boards with one processor, use the default property value 0 to specify the processor.

Given these two properties, the most common forms of the `ticcs` method are

```
cc = ticcs('boardnum',value)
cc = ticcs('boardnum',value,'procnum',value)
cc = ticcs(...,'timeout',value)
```

which specify the board, and processor in the second example, as the processor.

The third example adds the `timeout` input argument and `value` to allow you to specify how long MATLAB software waits for the connection to the processor or the response to a command to return completed.

Note The output argument name you provide for `ticcs` cannot begin with an underscore, such as `_cc`.

You do not need to specify the `boardnum` and `procnum` properties when you have one board with one processor installed. The default property values refer correctly to the processor on the board.

Note Simulators are considered boards. If you defined both boards and simulators in CCS IDE, specify the `boardnum` and `procnum` properties to connect to specific boards or simulators. Use `ccsboardinfo` to determine the values for the `boardnum` and `procnum` properties.

Because these properties are read only after you create the handle, you must set these property values as input arguments when you use `ticcs`. You cannot change these values after the handle exists. After you create the handle, use the `get` function to retrieve the `boardnum` and `procnum` property values.

Using ticcs with Multiple Processor Boards

When you create ticcs objects that access boards that contain more than one processor, such as the OMAP1510 platform, ticcs behaves a little differently.

For each of the ticcs syntaxes above, the result of the method changes in the multiple processor case, as follows.

```
cc = ticcs
cc = ticcs('propertyname',propertyvalue)
cc = ticcs('propertyname',propertyvalue,'propertyname',...
propertyvalue)
```

In the case where you do not specify a board or processor:

```
cc = ticcs
Array of TICCS Objects:
API version           : 1.2
Board name            : OMAP 3.0 Platform Simulator [Texas
Instruments]
Board number          : 0
Processor 0 (element 1): TMS470R2127 (MPU, Not Running)
Processor 1 (element 2): TMS320C5500 (DSP, Not Running)
```

Where you choose to identify your processor as an input argument to ticcs, for example, when your board contains two processors:

```
cc = ticcs('boardnum',2)
Array of TICCS Objects:
API version           : 1.2
Board name            : OMAP 3.0 Platform Simulator [Texas Instruments]
Board number          : 2
Processor 0 (element 1): TMS470R2127 (MPU, Not Running)
Processor 1 (element 2): TMS320C5500 (DSP, Not Running)
```

cc returns a two element object handle with cc(1) corresponding to the first processor and cc(2) corresponding to the second.

You can include both the board number and the processor number in the `ticcs` syntax, as shown here:

```
cc = ticcs('boardnum',2,'procnum',[0 1])
Array of TICCS Objects:
  API version           : 1.2
  Board name            : OMAP 3.0 Platform Simulator [Texas
  Instruments]
  Board number          : 2
  Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
  Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)
```

Enter `procnum` as either a single processor on the board (a single value in the input arguments to specify one processor) or a vector of processor numbers, as shown in the example, to select two or more processors.

Support Coemulation and OMAP

Coemulation, defined by Texas Instruments to mean simultaneous debugging of two or more CPUs, allows you to coordinate your debugging efforts between two or more processors within one device. Efficient development with OMAP™ hardware requires coemulation support. Instead of creating one `cc` object when you issue the following command

```
cc = ticcs
```

or your hardware that has multiple processors, the resulting `cc` object comprises a vector of `cc` objects `cc(1)`, `cc(2)`, and so on, each of which accesses one processor on your device, say an OMAP1510. When your processor has one processor, `cc` is a single object. With a multiprocessor board, the `cc` object returns the new vector of objects. For example, for board 2 with two processors,

```
cc = ticcs
```

returns the following information about the board and processors:

```
cc = ticcs('boardnum',2)
Array of TICCS Objects:
```



```

API version           : 1.2
Board name            : OMAP 3.0 Platform Simulator [Texas
Instruments]
Board number          : 2
Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)

```

Checking the existing boards shows that board 2 does have two processors:

```
ccsboardinfo
```

Board Num	Board Name	Proc Num	Processor Name	Processor Type
2	OMAP 3.0 Platform Simulator [T ...	0	MPU	TMS470R2x
2	OMAP 3.0 Platform Simulator [T ...	1	DSP	TMS320C550
1	MGS3 Simulator [Texas Instruments]	0	CPU	TMS320C5500
0	ARM925 Simulator [Texas Instru ...	0	CPU	TMS470R2x

Examples

On a system with three boards, where the third board has one processor and the first and second boards have two processors each, the following function:

```
cc = ticcs('boardnum',1,'procnum',0);
```

returns an object that accesses the first processor on the second board. Similarly, the function

```
cc = ticcs('boardnum',0,'procnum',1);
```

returns an object that refers to the second processor on the first board.

To access the processor on the third board, use

```
cc = ticcs('boardnum',2);
```

which sets the default property value `procnum= 0` to connect to the processor on the third board.

```
cc = ticcs
TICCS Object:
API version      : 1.2
Processor type   : TMS320C6711
Processor name   : CPU_1
Running?        : No
Board number     : 1
Processor number : 0
Default timeout  : 10.00 secs

RTDX channels    : 0

cc.type % Returns information about the type object

Defined types : Void, Float, Double, Long, Int, Short, Char
```

See Also

`ccsboardinfo`, `set`

Purpose	(For CCS) Set whether CCS IDE window is visible while CCS runs
Syntax	<code>visible(cc,state)</code>
Description	<p><code>visible(cc,state)</code> sets CCS IDE to be visible or not visible on the desktop. Input argument <code>state</code> accepts either 0 or 1 to set the visibility. Setting <code>state</code> equal to 0 makes CCS IDE not visible on the desktop. However, the CCS IDE process runs in the background while the window is not visible.</p> <p>Running CCS IDE without making it visible lets you use the CCS IDE functions from MATLAB software, without interacting with CCS IDE. If you need to interact with CCS IDE, set <code>state</code> equal to 1. This makes CCS IDE visible and you can use the features of the user window.</p> <p>An important feature of <code>visible</code> is that it creates a new link to CCS IDE when you change the IDE visibility. As a result, after you use</p> <pre>visible(cc,state)</pre> <p>to make CCS IDE show on your desktop, the MATLAB software <code>clear all</code> function does not remove the visibility handle. You must remove the handle explicitly before you use <code>clear</code>.</p> <p>To see the visibility difference, open CCS and use Microsoft Windows Task Manager to look at the applications and processes running on your computer. When CCS IDE is visible (the normal startup and operating mode for the IDE), CCS IDE appears listed on the Applications page of Task Manager. And the process <code>cc_app.exe</code> shows up on the Processes page as a running process. When you set CCS IDE to not visible (<code>state</code> equal 0), CCS IDE disappears from the Applications page, but remains on the Processes page, with a process ID (PID), using CPU and memory resources.</p>

Note When you close MATLAB software while CCS IDE is not visible, MATLAB software closes CCS if it started the IDE.

visible

For more information about visibility and CCS, refer to “Running Code Composer Studio Software on Your Desktop — Visibility” on page 2-5.

Examples

Test to see whether CCS IDE is running. Then change the visibility and check again. Start CCS IDE. Then open MATLAB software and at the prompt, enter

```
cc=ticcs;
```

MATLAB software creates a link to CCS IDE and leaves CCS IDE visible on your desktop.

```
isvisible(cc)  
  
ans =  
    1
```

Now, change the visibility state to 0, or invisible, and check the state.

```
visible(cc,0)  
isvisible(cc)  
  
ans =  
    0
```

Notice that CCS IDE is not visible on your desktop. Recall that MATLAB software did not open CCS IDE. When you close MATLAB software with CCS IDE in this invisible state, CCS IDE remains running in the background. To close it, do one of the following operations.

- Start MATLAB software. Create a new link to CCS IDE. Use the new link to make CCS IDE visible. Close CCS IDE.
- Open Microsoft Windows Task Manager. Click **Processes**. Find and highlight `cc_app.exe`. Click **End Task**.

See Also

`isvisible`, `load`

Purpose (For CCS) Write data to memory on processor

Syntax `write(cc,address,data,timeout)`
`write(cc,address,data)`

Description **ticcs Object Syntaxes**

`write(cc,address,data,timeout)` sends a block of data to memory on the processor referred to by `cc`. The `address` and `data` input arguments define the memory block to write—where the memory starts and what data is being written. The memory block to write to begins at the memory location defined by `address`. `data` is the data to write, and can be a scalar, a vector, a matrix, or a multidimensional array.

Data get written to memory in column-major order. `timeout` is an optional input argument you use to terminate long write processes and data transfers. For details about each input parameter, read the following descriptions.

To update values in memory on a running processor, such as values to change during processing, insert one or more breakpoints in the project code and perform the write operation while the processor code is paused at one of the breakpoints. After you read the data, release the breakpoint.

Note

Do not attempt to write data to the processor while it is running. Writing data to a running process can result in incorrect data in memory or in program use.

`address` — `write` uses `address` to define the beginning of the memory block to write to. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector [`location`, `page`], a string, or a decimal value.

When the processor has only one memory page, as is true for many digital signal processors, the value of the page portion of the memory address is 0. By default, `ticcs` sets the page value to 0 at creation if you omit `page` as an input argument.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
1F	String	Offset is 31 decimal on the page referred to by <code>cc.page</code>
10	Decimal	Offset is 10 decimal on the page referred to by <code>cc.page</code>
[18,1]	Vector	Offset is 18 decimal on memory page 1 (<code>cc.page = 1</code>)

To specify the address in hexadecimal format, enter the address property value as a string. `write` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the `write` uses `hex2dec`. When you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc.page`.

`data` — the scalar, vector, or array of values that are written to memory on the processor. `write` supports the following data types:

Datatypes	Description
<code>double</code>	Double-precision floating point values
<code>int8</code>	Signed 8-bit integers

Datatypes	Description
int16	Signed 16-bit integers
int32	Signed 32-bit integers
single	Single-precision floating point data
uint8	Unsigned 8-bit integers
uint16	Unsigned 16-bit integers
uint32	Unsigned 32-bit integers

To limit the time that `write` spends transferring data from the processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You may find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, `write` defaults to the global timeout defined in `cc`.

`write(cc, address, data)` sends a block of data to memory on the processor referred to by `cc`. The `address` and `data` input arguments define the memory block to be written—where the memory starts and what data is being written. The memory block to be written to begins at the memory location defined by `address`. `data` is the data to be written, and can be a scalar, a vector, a matrix, or a multidimensional array.

Data get written to memory in column-major order. Refer to the preceding syntax for details about the input arguments. In this syntax, `timeout` defaults to the global timeout period defined in `cc.timeout`. Use `get` to determine the default `timeout` value.

Like the `isreadable`, `iswritable`, and `read` functions, `write` checks for valid address values. Illegal address values would be any address space larger than the available space for the processor – 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

Writing Negative Values

Writing a negative value causes the data written to be saturated because char is unsigned on the processor. Hence, a 0 (a NULL) is written instead. A warning results as well, as this example shows.

```
cc = ticcs;
ff = createobj(cc,'g_char'); % Where g_char is in the code.
write(ff,-100);
Warning: Underflow: Saturation was required to fit the data into
an addressable unit.
```

When you try to read the data you wrote, the character being read is 0 (NULL) — so there seems to be nothing returned by the read function.

You can demonstrate this by the following code, after writing a negative value as shown in the previous example.

```
readnumeric(x)
ans =
0
read(x) % Reads the NULL character.
ans = % Apparently nothing is returned.

double(read(x)) % Read the numeric equivalent of NULL.
ans = % Again, appears not to return a value.
```

Examples

The following examples demonstrate how to use write. cc is a ticcs object.

Connect to a processor and write data to it. In this example, CCS IDE recognizes one board having one processor.

```
cc = ticcs;
cc.visible(1);
write(cc,'50',1:250);
mem = read(cc,0,'double',50) % Returns 50 values as a column
                             % vector in mem.
```


It may be more convenient to return the data in an array. If you enter a vector for `count`, `mem` contains a matrix of dimensions the same as vector `count`.

```
write(cc,10,1:100);
mem=read(cc,10,'double',[10 10])
```

```
mem =
```

```

 1  11  21  31  41  51  61  71  81  91
 2  12  22  32  42  52  62  72  82  92
 3  13  23  33  43  53  63  73  83  93
 4  14  24  34  44  54  64  74  84  94
 5  15  25  35  45  55  65  75  85  95
 6  16  26  36  46  56  66  76  86  96
 7  17  27  37  47  57  67  77  87  97
 8  18  28  38  48  58  68  78  88  98
 9  19  29  39  49  59  69  79  89  99
10  20  30  40  50  60  70  80  90 100
```

Write an array of 16-bit integers at the location of target symbol data.

```
write(cc,address(h,'data'),int16([1:100]))
```

Write a single-precision, IEEE floating-point value (32-bits) at address FF00(Hex).

```
write(cc,'FF00',single(23.5))
```

Write a 2-D array of integers in row-major (C-style) format at address 65280 (decimal).

```
mlarr = int32([1:10; 101:110])
write(cc,65280,mlarr')
```

See Also

`read`, `symbol`

writemsg

Purpose

(For CCS) Write messages to specified RTDX channel

Note Support for writemsg on C5000 and C6000 processors will be removed in a future version.

Syntax

```
data = writemsg(rx,channelname,data)
data = writemsg(rx,channelname,data)
```

Description

`data = writemsg(rx,channelname,data)` writes `data` to a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for write access. All messages must be the same type for a single write operation. `writemsg` takes the elements of matrix `data` in column-major order.

To limit the time that `writemsg` spends transferring messages from the processor, the optional argument `timeout` tells the message transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You may find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, write defaults to the global timeout period defined in `cc`.

`writemsg` supports the following data types: `uint8`, `int16`, `int32`, `single`, and `double`.

`data = writemsg(rx,channelname,data)` uses the global timeout setting assigned to `cc` when you create the link.

Examples

After you load a program to your processor, configure a link in RTDX for write access and use `writemsg` to write data to the processor. Recall that the program loaded on the processor must define `ichannel` and the channel must be configured for write access.

```
cc=ticcs;
rx = cc.rtdx;
open(rx,'ichannel','w'); % Could use rx.open('ichannel','w')
```

```
enable(rx, 'ichannel');  
inputdata(1:25);  
writemsg(rx, 'ichannel', int16(inputdata));
```

As a further illustration, the following code snippet writes the messages in matrix `indata` to the write-enabled channel specified by `ichan`. Note again that this example works only when `ichan` is defined by the program on the processor and enabled for write access.

```
indata = [1 4 7; 2 5 8; 3 6 9];  
writemsg(cc.rtdx, 'ichan', indata);
```

The matrix `indata` is written by column to `ichan`. The preceding function syntax is equivalent to

```
writemsg(cc.rtdx, 'ichan', [1:9]);
```

See Also

`readmat`, `readmsg`, `write`

writemsg

Block Reference

Block Library: idelinklib_ticcs

C280x/C2802x/C2803x/C28x3x
Hardware Interrupt

Interrupt Service Routine to
handle hardware interrupt on
C280x/C28x3x processors

C281x Hardware Interrupt

Interrupt Service Routine to handle
hardware interrupt

C5000/C6000 Hardware Interrupt

Interrupt Service Routine to handle
hardware interrupt on C5000 and
C6000 processors

Block Library: idelinklib_common

Idle Task

Memory Allocate

Memory Copy

Create free-running task

Allocate memory section

Copy to and from memory section

Blocks — Alphabetical List

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

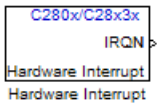
Purpose

Interrupt Service Routine to handle hardware interrupt on C280x/C28x3x processors

Library

Embedded IDE Link for TI Code Composer Studio (idelinklib_ticcs)

Description



For many systems, an execution scheduling model based on a timer interrupt is not sufficient to ensure a real-time response to external events. The C280x/C28x3x Hardware Interrupt block addresses this problem by allowing asynchronous processing of interrupts triggered by events managed by other blocks in the C280x/C28x3x DSP Chip Support Library.

The following C280x/C28x3x blocks that can generate an interrupt for asynchronous processing are available in Target Support Package.

- C280x ADC
- C280x eCAN Receive
- C280x SCI Receive
- C280x SCI Transmit
- C280x SPI Receive
- C280x SPI Transmit

Only one Hardware Interrupt block can be used in a model. To handle multiple interrupts, place a Demux block at the output of the Hardware Interrupt block to direct function calls to the appropriate function-call subsystems.

Vectorized Output

The output of this block is a function call. The size of the function call line equals the number of interrupts the block is set to handle. Each interrupt is represented by four parameters shown on the dialog box of the block. These parameters are a set of four vectors of equal length. Each interrupt is represented by one element from each parameter (four elements total), one from the same position in each of these vectors.

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

Each interrupt is described by:

- CPU interrupt numbers
- PIE interrupt numbers
- Task priorities
- Preemption flags

So one interrupt is described by a CPU interrupt number, a PIE interrupt number, a task priority, and a preemption flag.

The CPU and PIE interrupt numbers together uniquely specify a single interrupt for a single peripheral or peripheral module. The following table maps CPU and PIE interrupt numbers to these peripheral interrupts.

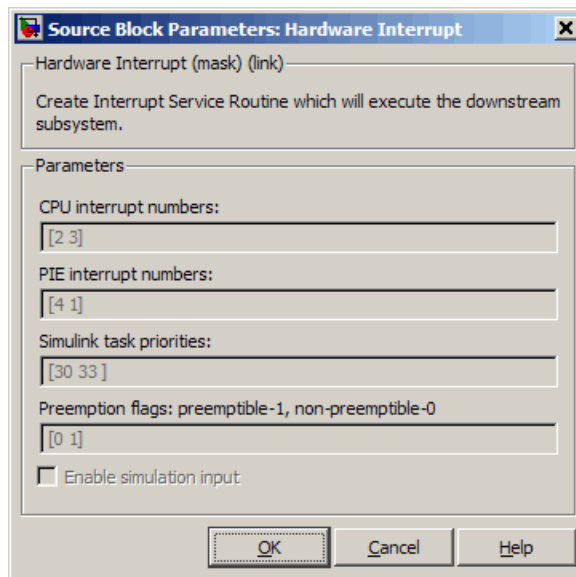
C280x Peripheral Interrupt Vector Values

Row numbers = CPU values/Column numbers = PIE values								
	8	7	6	5	4	3	2	1
1	WAKEINT (LPM/WD)	TINT0 (TIMER 0)	ADCINT (ADC)	XINT2	XINT1	Reserved	SEQ2INT (ADC)	SEQ1INT (ADC)
2	Reserved	Reserved	EPWM6_TZINT (ePWM6)	EPWM5_TZINT (ePWM5)	EPWM4_TZINT (ePWM4)	EPWM3_TZINT (ePWM3)	EPWM2_TZINT (ePWM2)	EPWM1_TZINT (ePWM1)
3	Reserved	Reserved	EPWM6_INT (ePWM6)	EPWM5_INT (ePWM5)	EPWM4_INT (ePWM4)	EPWM3_INT (ePWM3)	EPWM2_INT (ePWM2)	EPWM1_INT (ePWM1)
4	Reserved	Reserved	Reserved	Reserved	ECAP4_INT (eCAP4)	ECAP3_INT (eCAP3)	ECAP2_INT (eCAP2)	ECAP1_INT (eCAP1)
5	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	EQEP2_INT (eQEP2)	EQEP1_INT (eQEP1)
6	SPITXINTD (SPI-D)	SPIRXINTD (SPI-D)	SPITXINTC (SPI-C)	SPIRXINTC (SPI-C)	SPITXINTB (SPI-B)	SPIRXINTB (SPI-B)	SPITXINTA (SPI-A)	SPIRXINTA (SPI-A)
7	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
8	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	I2CINT1A (I2C-A)	I2CINT2A (I2C-A)
9	ECAN1INTB (CAN-B)	ECAN0INTB (CAN-B)	ECAN1INTA (CAN-A)	ECAN0INTA (CAN-A)	SCITXINTB (SCI-B)	SCIRXINTB (SCI-B)	SCITXINTA (SCI-A)	SCIRXINTA (SCI-A)
10	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
11	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
12	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

The task priority indicates the relative importance tasks associated with the asynchronous interrupts. If an interrupt triggers a higher-priority task while a lower-priority task is running, the execution of the lower-priority task will be suspended while the higher-priority task is executed. The lowest value represents the highest priority. The default priority value of the base rate task is 40, so the priority value for each asynchronously triggered task must be less than 40 for these tasks to suspend the base rate task.

The preemption flag determines whether a given interrupt is preemptable. Preemption overrides prioritization, such that a preemptable task of higher priority can be preempted by a non-preemptable task of lower priority.

Dialog Box



CPU interrupt numbers

Enter a vector of CPU interrupt numbers for the interrupts you want to process asynchronously.

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

See the table of C280x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

PIE interrupt numbers

Enter a vector of PIE interrupt numbers for the interrupts you want to process asynchronously.

See the table of C280x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

Simulink task priorities

Enter a vector of task priorities for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 9-2 for an explanation of task priorities.

Preemption flags

Enter a vector of preemption flags for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 9-2 for an explanation of preemption flags.

Enable simulation input

Select this check box if you want to be able to test asynchronous interrupt processing in the context of your Simulink software model.

Note Select this check box to enable you to test asynchronous interrupt processing behavior in Simulink software.

References

Detailed information about interrupt processing is in *TMS320x280x DSP System Control and Interrupts Reference Guide*, Literature Number SPRU712B, available at the Texas Instruments Web site.

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

See Also

The following links refer to block reference pages that require the Target Support Package software.

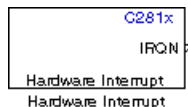
[C280x/C28x3x/C2802x Software Interrupt Trigger, Idle Task](#)

C281x Hardware Interrupt

Purpose Interrupt Service Routine to handle hardware interrupt

Library Embedded IDE Link for TI Code Composer Studio (idelinklib_ticcs)

Description



For many systems, an execution scheduling model based on a timer interrupt is not sufficient to ensure a real-time response to external events. The C281x Hardware Interrupt block addresses this problem by allowing for the asynchronous processing of interrupts triggered by events managed by other blocks in the C281x DSP Chip Support Library.

The following C281x blocks that can generate an interrupt for asynchronous processing are available from Target Support Package:

- C281x ADC
- C281x CAP
- C281x eCAN Receive
- C281x Timer
- C281x SCI Receive
- C281x SCI Transmit
- C281x SPI Receive
- C281x SPI Transmit

Only one Hardware Interrupt block can be used in a model. To handle multiple interrupts, place a Demux block at the output of the Hardware Interrupt block to direct function calls to the appropriate function-call subsystems.

Vectorized Output

The output of this block is a function call. The size of the function call line equals the number of interrupts the block is set to handle. Each interrupt is represented by four parameters shown on the dialog box of the block. These parameters are a set of four vectors of equal length.

Each interrupt is represented by one element from each parameter (four elements total), one from the same position in each of these vectors.

Each interrupt is described by:

- CPU interrupt numbers
- PIE interrupt numbers
- Task priorities
- Preemption flags

So one interrupt is described by a CPU interrupt number, a PIE interrupt number, a task priority, and a preemption flag.

The CPU and PIE interrupt numbers together uniquely specify a single interrupt for a single peripheral or peripheral module. The following table maps CPU and PIE interrupt numbers to these peripheral interrupts.

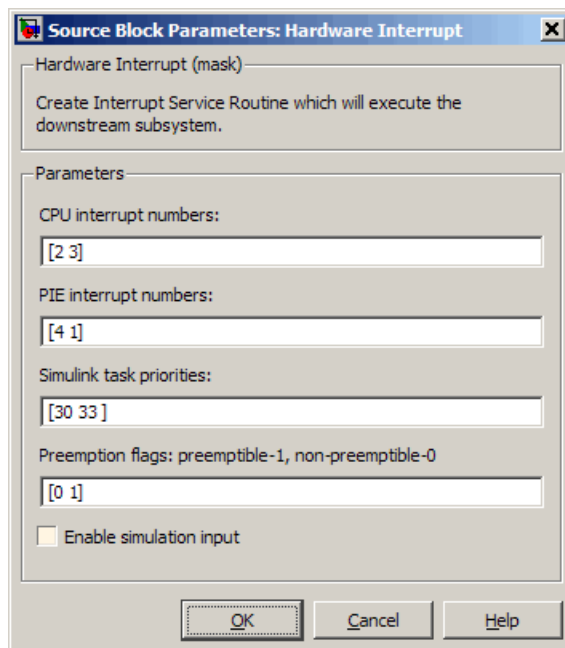
C281x Peripheral Interrupt Vector Values

Row numbers = CPU values / Column numbers = PIE values								
	8	7	6	5	4	3	2	1
1	WAKEINT (LPM/WMD)	TINT0 (TIMER 0)	ADCINT (ADC)	XINT2	XINT1	Reserved	PDPINTB (EV-B)	PDPINTA (EV-A)
2	Reserved	T1OFINT (EV-A)	T1UFINT (EV-A)	T1CINT (EV-A)	T1PINT (EV-A)	CMP3INT (EV-A)	CMP2INT (EV-A)	CMP1INT (EV-A)
3	Reserved	CAPINT3 (EV-A)	CAPINT2 (EV-A)	CAPINT1 (EV-A)	T2OFINT (EV-A)	T2UFINT (EV-A)	T2CINT (EV-A)	T2PINT (EV-A)
4	Reserved	T3OFINT (EV-B)	T3UFINT (EV-B)	T3CINT (EV-B)	T3PINT (EV-B)	CMP6INT (EV-B)	CMP5INT (EV-B)	CMP4INT (EV-B)
5	Reserved	CAPINT6 (EV-B)	CAPINT5 (EV-B)	CAPINT4 (EV-B)	T4OFINT (EV-B)	T4UFINT (EV-B)	T4CINT (EV-B)	T4PINT (EV-B)
6	Reserved	Reserved	MXINT (McBSP)	MRINT (McBSP)	Reserved	Reserved	SPITXINTA (SPI)	SPIRXINTA (SPI)
7	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
8	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
9	Reserved	Reserved	ECAN1INT (CAN)	ECAN0INT (CAN)	SCITXINTB (SCI-B)	SCIRXINTB (SCI-B)	SCITXINTA (SCI-A)	SCIRXINTA (SCI-A)
10	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
11	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
12	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

The task priority indicates the relative importance tasks associated with the asynchronous interrupts. If an interrupt triggers a higher-priority task while a lower-priority task is running, the execution of the lower-priority task will be suspended while the higher-priority task is executed. The lowest value represents the highest priority. Note that the default priority value of the base rate task is 40, so the priority value for each asynchronously triggered task must be less than 40 for these tasks to actually cause the suspension of the base rate task.

The preemption flag determines whether a given interrupt is preemptable or not. Preemption overrides prioritization, such that a preemptable task of higher priority can be preempted by a non-preemptable task of lower priority.

Dialog Box



C281x Hardware Interrupt

CPU interrupt numbers

Enter a vector of CPU interrupt numbers for the interrupts you want to process asynchronously.

See the table of C281x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

PIE interrupt numbers

Enter a vector of PIE interrupt numbers for the interrupts you want to process asynchronously.

See the table of C281x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

Simulink task priorities

Enter a vector of task priorities for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 9-8 for an explanation of task priorities.

Preemption flags

Enter a vector of preemption flags for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 9-8 for an explanation of preemption flags.

Enable simulation input

Select this check box if you want to be able to test asynchronous interrupt processing in the context of your Simulink software model.

Note Use this check box to enable you to test asynchronous interrupt processing behavior in Simulink software.

References

Detailed information interrupt processing is in *TMS320x281x DSP System Control and Interrupts Reference Guide*, Literature Number SPRU078C, available at the Texas Instruments Web site.

See Also

The following links to block reference pages require that Target Support Package is installed.

C281x Software Interrupt Trigger, C281x Timer, Idle Task

C5000/C6000 Hardware Interrupt

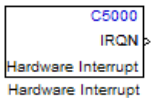
Purpose

Interrupt Service Routine to handle hardware interrupt on C5000 and C6000 processors

Library

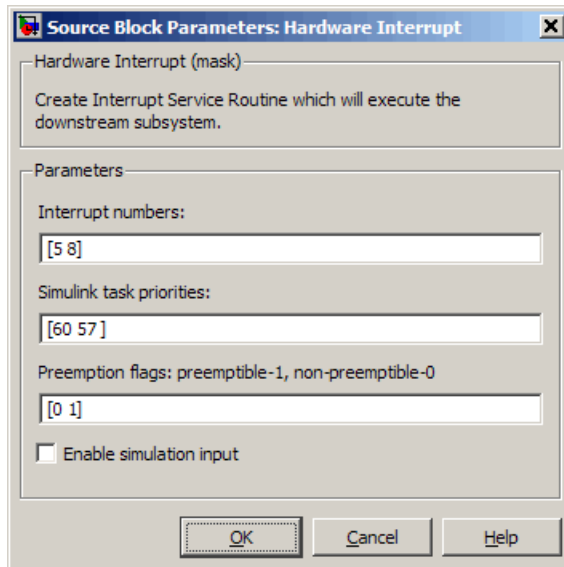
Embedded IDE Link for TI Code Composer Studio (idelinklib_ticcs)

Description



Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that run the processes that are downstream from the this block or a Task block connected to this block.

Dialog Box



Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The following table provides the valid range for C5xxx and C6xxx processors:

C5000/C6000 Hardware Interrupt

Processor Family	Valid Interrupt Numbers
C5xxx	2, 3, 5-21, 23
C6xxx	4-15

The width of the block output signal corresponds to the number of interrupt numbers specified here. Combined with the **Simulink task priorities** that you enter and the preemption flag you enter for each interrupt, these three values define how the code and processor handle interrupts during asynchronous scheduler operations.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink software task priority specifies the Simulink priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Simulink task priority values are required to generate the proper rate transition code (refer to Rate Transitions and Asynchronous Blocks). The task priority values are also required to ensure absolute time integrity when the asynchronous task needs to obtain real time from its base rate or its caller. Typically, you assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

Preemption flags preemptable – 1, non-preemptable – 0

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted.
Entering 0 indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of

C5000/C6000 Hardware Interrupt

the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 5 in **Interrupt numbers** is not preemptible and the priority 8 interrupt can be preempted.

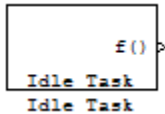
Enable simulation input

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Purpose Create free-running task

Library Block Library: idelinklib_common

Description



The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. All tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

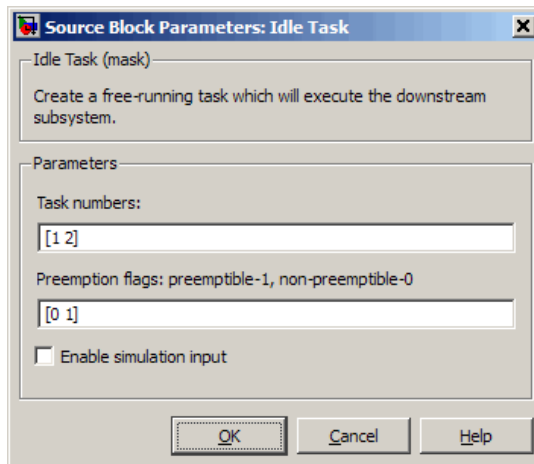
Vectorized Output

The block output comprises a set of vectors—the task numbers vector and the preemption flag or flags vector. Any preemption-flag vector must be the same length as the number of tasks vector unless the preemption flag vector has only one element. The value of the preemption flag determines whether a given interrupt (and task) is preemptible. Preemption overrides prioritization. A lower-priority nonpreemptible task can preempt a higher-priority preemptible task.

When the preemption flag vector has one element, that element value applies to all functions in the downstream subsystem as defined by the task numbers in the task number vector. If the preemption flag vector has the same number of elements as the task number vector, each task defined in the task number vector has a preemption status defined by the value of the corresponding element in the preemption flag vector.

Idle Task

Dialog Box



Task numbers

Identifies the created tasks by number. Enter as many tasks as you need by entering a vector of integers. The default values are [1, 2] to indicate that the downstream subsystem has two functions.

The values you enter determine the execution order of the functions in the downstream subsystem, while the number of values you enter corresponds to the number of functions in the downstream subsystem.

Enter a vector containing the same number of elements as the number of functions in the downstream subsystem. This vector can contain no more than 16 elements, and the values must be from 0 to 15 inclusive.

The value of the first element in the vector determines the order in which the first function in the subsystem is executed, the value of the second element determines the order in which the second function in the subsystem is executed, and so on.

For example, entering [2,3,1] in this field indicates that there are three functions to be executed, and that the third function is executed first, the first function is executed second, and the second function is executed third. After all functions are executed, the Idle Task block cycles back and repeats the execution of the functions in the same order.

Preemption flags

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Task numbers** contains more than one task, you can assign different preemption flags to each task by entering a vector of flag values, corresponding to the order of the tasks in **Task numbers**. If **Task numbers** contains more than one task, and you enter only one flag value here, that status applies to all tasks.

In the default settings [0 1], the task with priority 1 in **Task numbers** is not preemptible, and the priority 2 task can be preempted.

Enable simulation input

When you select this option, Simulink software adds an input port to the Idle Task block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

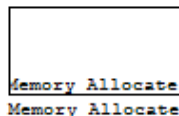
Note Select this check box to test asynchronous interrupt processing behavior in Simulink software.

Memory Allocate

Purpose Allocate memory section

Library Block Library: idelinklib_common

Description On C2xxx, C5xxx, or C6xxx processors, this block directs the TI compiler to allocate memory for a new variable you specify. Parameters in the block dialog box let you specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.



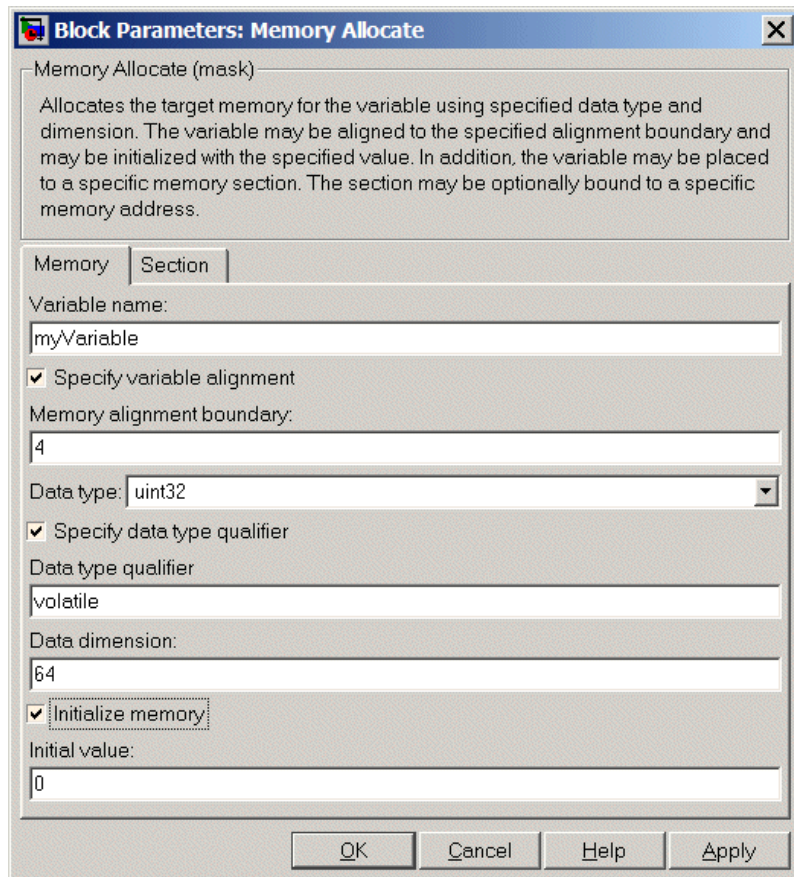
The block does not verify whether the entries for your variable are valid, such as checking the variable name, data type, or section. You must ensure that all variable names are valid, that they use valid data types, and that all section names you specify are valid as well.

The block does not have input or output ports. It only allocates a memory location. You do not connect it to other blocks in your model.

Dialog Box The block dialog box comprises multiple tabs:

- **Memory** — Allocate the memory for storing variables. Specify the data type and size.
- **Section** — Specify the memory section in which to allocate the variable.

The dialog box images show all of the available parameters enabled. Some of the parameters shown do not appear until you select one or more other parameters.



The following sections describe the contents of each pane in the dialog box.

Memory Allocate

Memory Parameters

Block Parameters: Memory Allocate

Memory Allocate (mask)

Allocates the target memory for the variable using specified data type and dimension. The variable may be aligned to the specified alignment boundary and may be initialized with the specified value. In addition, the variable may be placed to a specific memory section. The section may be optionally bound to a specific memory address.

Memory | Section

Variable name:
myVariable

Specify variable alignment

Memory alignment boundary:
4

Data type: uint32

Specify data type qualifier

Data type qualifier
volatile

Data dimension:
64

Initialize memory

Initial value:
0

OK Cancel Help Apply

You find the following memory parameters on this tab.

Variable name

Specify the name of the variable to allocate. The variable is allocated in the generated code.

Specify variable alignment

Select this option to direct the compiler to align the variable in **Variable name** to an alignment boundary. When you select this option, the **Memory alignment boundary** parameter appears so you can specify the alignment. Use this parameter and **Memory alignment boundary** when your processor requires this feature.

Memory alignment boundary

After you select **Specify variable alignment**, this option enables you to specify the alignment boundary in bytes. If your variable contains more than one value, such as a vector or an array, the elements are aligned according to rules applied by the compiler.

Data type

Defines the data type for the variable. Select from the list of types available.

Specify data type qualifier

Selecting this enables **Data type qualifier** so you can specify the qualifier to apply to your variable.

Data type qualifier

After you select **Specify data type qualifier**, you enter the desired qualifier here. `Volatile` is the default qualifier. Enter the qualifier you need as text. Common qualifiers are `static` and `register`. The block does not check for valid qualifiers.

Data dimension

Specifies the number of elements of the type you specify in **Data type**. Enter an integer here for the number of elements.

Initialize memory

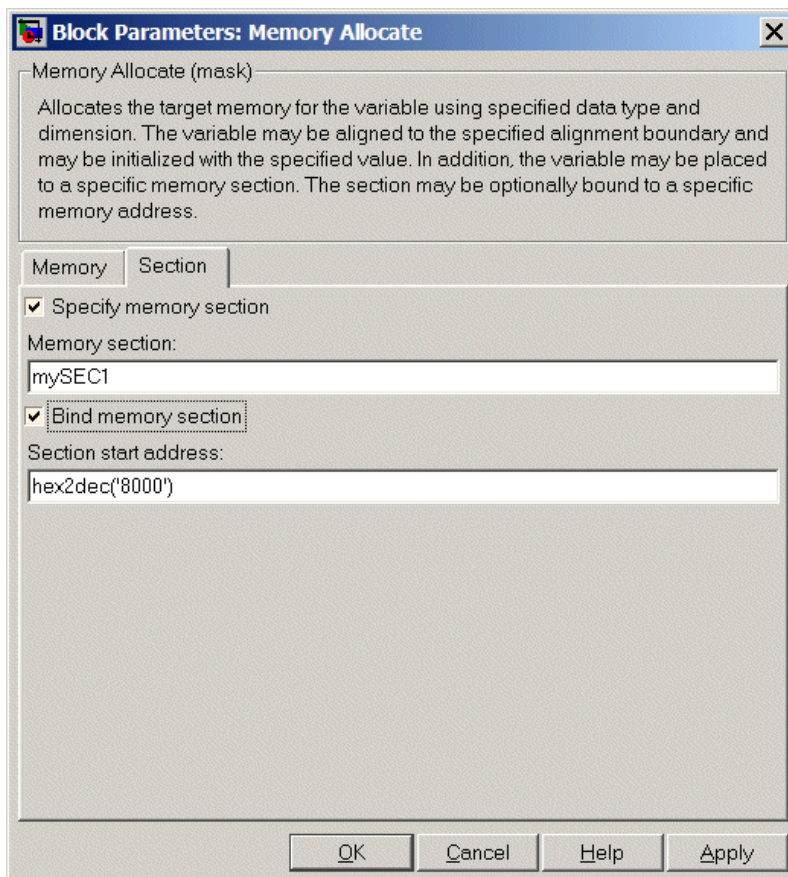
Directs the block to initialize the memory location to a fixed value before processing.

Initial value

Specifies the initialization value for the variable. At run time, the block sets the memory location to this value.

Memory Allocate

Section Parameters



Parameters on this pane specify the section in memory to store the variable.

Specify memory section

Selecting this parameter enables you to specify the memory section to allocate space for the variable. Enter either one of the

standard memory sections or a custom section that you declare elsewhere in your code.

Memory section

Identify a specific memory section to allocate the variable in **Variable name**. Verify that the section has sufficient space to store your variable. After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use **Bind memory section** to bind the memory section to a location.

Bind memory section

After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use this parameter to bind the memory section to the location in memory specified in **Section start address**. When you select this, you enable the **Section start address** parameter.

The new memory section specified in **Memory section** is defined when you check this parameter.

Note Do not use **Bind memory section** for existing memory sections.

Section start address

Specify the address to which to bind the memory section. Enter the address in decimal form or in hexadecimal with a conversion to decimal as shown by the default value `hex2dec('8000')`. The block does not verify the address—verify that the address exists and can contain the memory section you entered in **Memory section**.

See Also

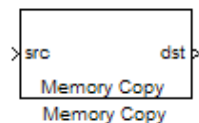
Memory Copy

Memory Copy

Purpose Copy to and from memory section

Library Block Library: idelinklib_common

Description



In generated code, this block copies variables or data from and to processor memory as configured by the block parameters. Your model can contain as many of these blocks as you require to manipulate memory on your processor.

Each block works with one variable, address, or set of addresses provided to the block. Parameters for the block let you specify both the source and destination for the memory copy, as well as options for initializing the memory locations.

Using parameters provided by the block, you can change options like the memory stride and offset at run time. In addition, by selecting various parameters in the block, you can write to memory at program initialization, at program termination, and at every sample time. The initialization process occurs once, rather than occurring for every read and write operation.

With the custom source code options, the block enables you to add custom ANSI C source code before and after each memory read and write (copy) operation. You can use the custom code capability to lock and unlock registers before and after accessing them. For example, some processors have registers that you may need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

If your processor or board supports quick direct memory access (QDMA) the block provides a parameter to check that implements the QDMA copy operation, and enables you to specify a function call that can indicate that the QDMA copy is finished. Only the C621x, C64xx, and C671x processor families support QDMA copy.

Block Operations

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. With the options for setting memory initialization and termination, you

control when and how the block initializes memory, copies to and from memory, and terminates memory operations. The parameters enable you to turn on and off memory operations in all three periods independently.

Used in combination with the Memory Allocate block, this block supports building custom device drivers, such as PCI bus drivers or codec-style drivers, by letting you manipulate and allocate memory. This block does not require the Memory Allocate block to be in the model.

In a simulation, this block does not perform any operation. The block output is not defined.

Copy Memory

When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

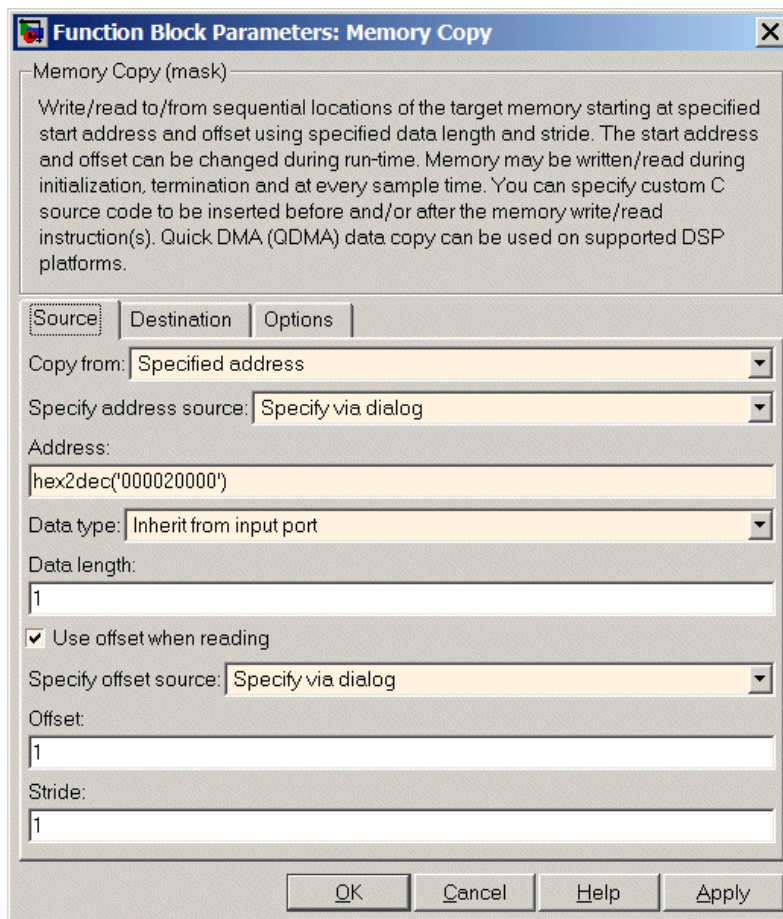
Dialog Box

The block dialog box contains multiple tabs:

- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various parameters to control the copy process.

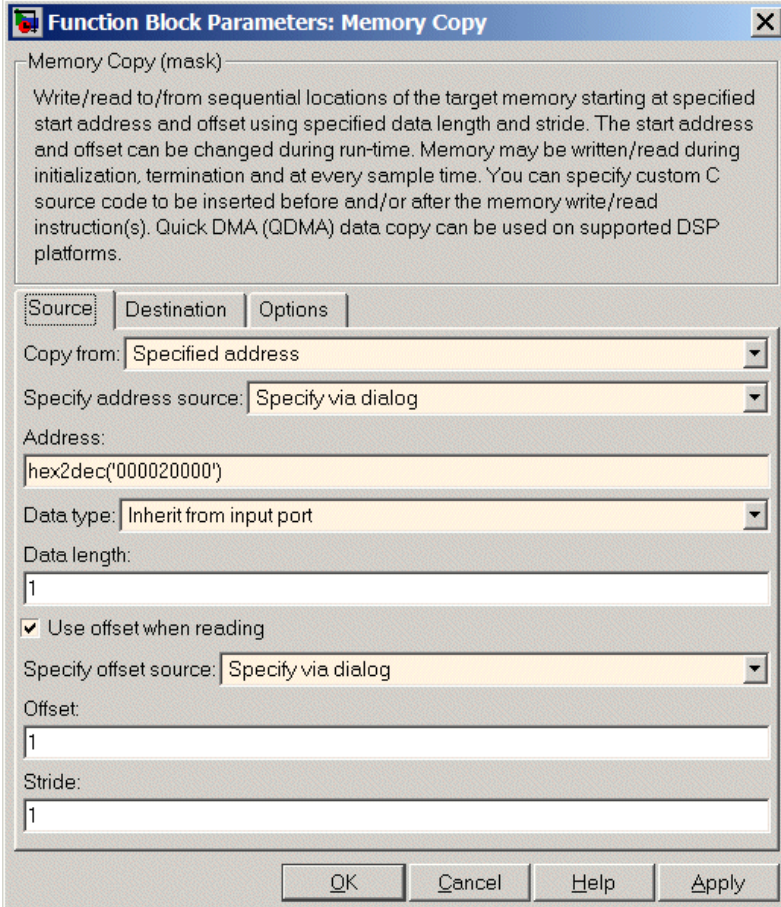
The dialog box images show many of the available parameters enabled. Some parameters shown do not appear until you select one or more other parameters. Some parameters are not shown in the figures, but the text describes them and how to make them available.

Memory Copy



Sections that follow describe the parameters on each tab in the dialog box.

Source Parameters



The image shows a dialog box titled "Function Block Parameters: Memory Copy". It contains a text area with a description of the block's function, followed by several input fields and checkboxes for configuring the memory copy operation. The fields include "Copy from", "Specify address source", "Address", "Data type", "Data length", "Use offset when reading", "Specify offset source", "Offset", and "Stride".

Memory Copy (mask)

Write/read to/from sequential locations of the target memory starting at specified start address and offset using specified data length and stride. The start address and offset can be changed during run-time. Memory may be written/read during initialization, termination and at every sample time. You can specify custom C source code to be inserted before and/or after the memory write/read instruction(s). Quick DMA (QDMA) data copy can be used on supported DSP platforms.

Source Destination Options

Copy from: Specified address

Specify address source: Specify via dialog

Address:
hex2dec('000020000')

Data type: Inherit from input port

Data length:
1

Use offset when reading

Specify offset source: Specify via dialog

Offset:
1

Stride:
1

OK Cancel Help Apply

Copy from

Select the source of the data to copy. Choose one of the entries on the list:

- **Input port** — This source reads the data from the block input port.

- Specified address — This source reads the data at the specified location in **Specify address source** and **Address**.
- Specified source code symbol — This source tells the block to read the symbol (variable) you enter in **Source code symbol**. When you select this copy from option, you enable the **Source code symbol** parameter.

Note If you do not select **Input port** for **Copy from**, change **Data type** from the default **Inherit from source** to one of the data types on the **Data type** list. If you do not make the change, you receive an error message that the data type cannot be inherited because the input port does not exist.

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either **Specify via dialog** or **Input port** from the list. Selecting **Specify via dialog** activates the **Address** parameter for you to enter the address for the variable.

When you select **Input port**, the port label on the block changes to **&src**, indicating that the block expects the address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax. Enter a string to specify the symbol exactly as you use it in your code.

Address

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use `hex2dec` to convert the address to the proper format. The following example converts `0x1000` to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you could enter either `4096` or `hex2dec('1000')` as the address.

Data type

Use this parameter to specify the type of data that your source uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `Inherit from source` for inheriting the data type from the block input port.

Data length

Specifies the number of elements to copy from the source location. Each element has the data type specified in **Data type**.

Use offset when reading

When you are reading the input, use this parameter to specify an offset for the input read. The offset value is in elements with the assigned data type. The **Specify offset source** parameter becomes available when you check this option.

Specify offset source

The block provides two sources for the offset — `Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when reading the input data.

Memory Copy

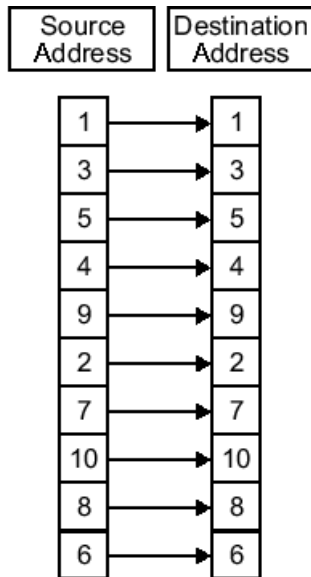
Offset

Offset tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the default value and **Offset** accepts only positive integers of one or greater.

Stride

Stride lets you specify the spacing for reading the input. By default, the stride value is one, meaning the generated code reads the input data sequentially. When you add a stride value that is not equal to one, the block reads the input data elements not sequentially, but by skipping spaces in the source address equal to the stride. **Stride** must be a positive integer.

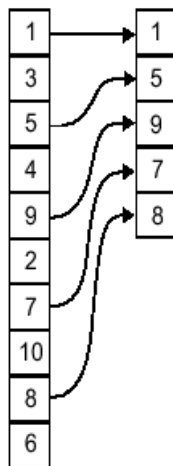
The next two figures help explain the stride concept. In the first figure you see data copied without any stride. Following that figure, the second figure shows a stride value of two applied to reading the input when the block is copying the input to an output location. You can specify a stride value for the output with parameter **Stride** on the **Destination** pane. Compare stride with offset to see the differences.



Input Stride = 1
Output Stride = 1
Number of Elements Copied = 10

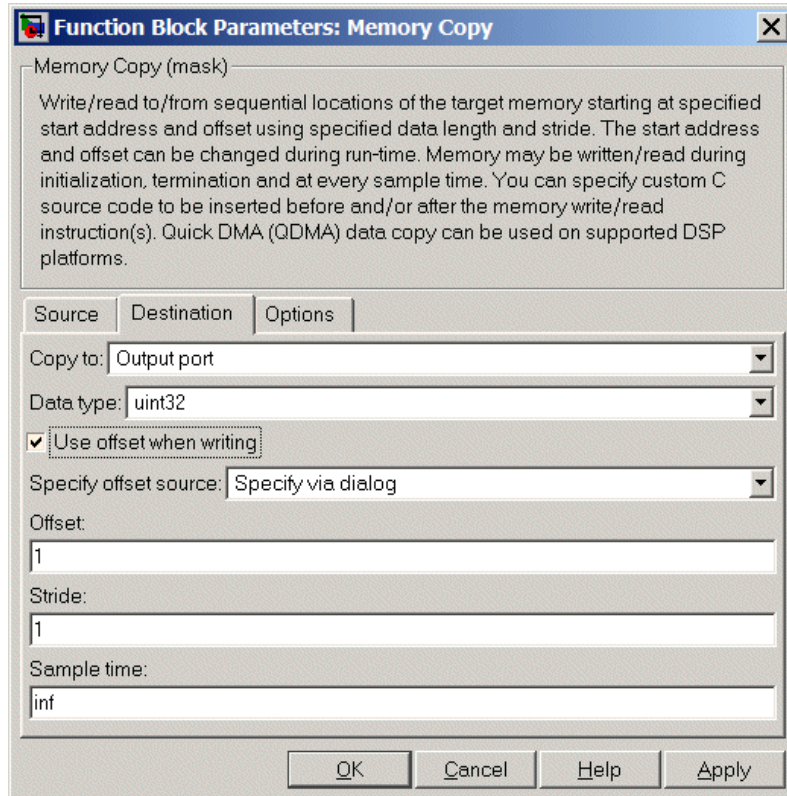
Memory Copy

Source Address	Destination Address
----------------	---------------------



Input Stride = 2
Output Stride = 1
Number of Elements Copied = 5

Destination Parameters



Copy to

Select the destination for the data. Choose one of the entries on the list:

- **Output port** — Copies the data to the block output port. From the output port the block passes data to downstream blocks in the code.
- **Specified address** — Copies the data to the specified location in **Specify address source** and **Address**.

- **Specified source code symbol** — Tells the block to copy the variable or symbol (variable) to the symbol you enter in **Source code symbol**. When you select this copy to option, you enable the **Source code symbol** parameter.

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either **Specify via dialog** or **Input port** from the list. Selecting **Specify via dialog** activates the **Address** parameter for you to enter the address for the variable.

When you select **Input port**, the port label on the block changes to **&dst**, indicating that the block expects the destination address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax.

Address

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use `hex2dec` to convert the address to the proper format. This example converts `0x2000` to decimal form.

```
8192 = hex2dec('2000');
```

For this example, you could enter either 8192 or `hex2dec('2000')` as the address.

Data type

Use this parameter to specify the type of data that your variable uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `inherit from source` for inheriting the data type for the variable from the block input port.

Specify offset source

The block provides two sources for the offset—`Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when writing the output data.

Offset

Offset tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before writing the first value to the destination. One is the default offset value and **Offset** accepts only positive integers of one or greater.

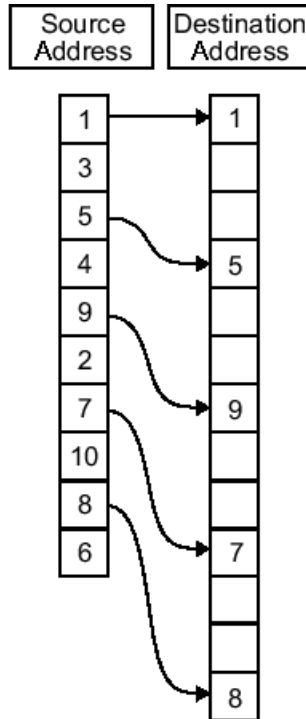
Stride

Stride lets you specify the spacing for copying the input to the destination. By default, the stride value is one, meaning the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value not equal to one, the output data is stored not sequentially, but by skipping addresses equal to the stride. **Stride** must be a positive integer.

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in

Memory Copy

the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.



Input Stride = 2
Output Stride = 3
Number of Elements Copied = 5

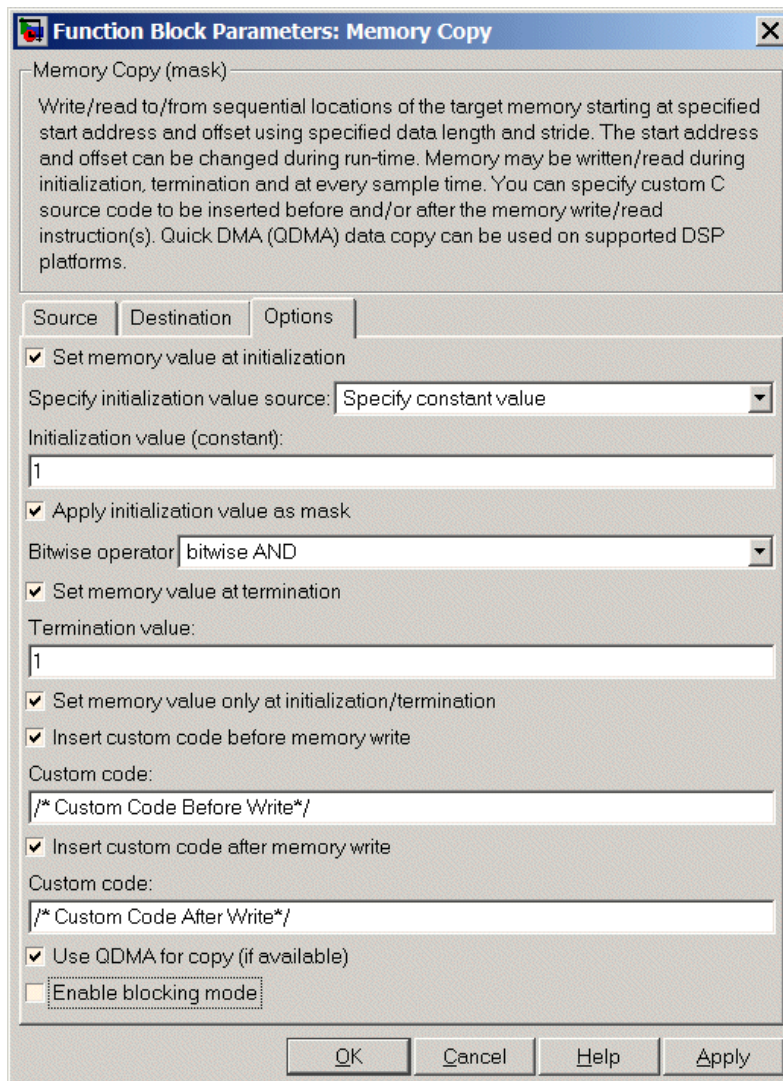
Sample time

Sample time sets the rate at which the memory copy operation occurs, in seconds. The default value Inf tells the block to use a constant sample time. You can set **Sample time** to -1 to direct the block to inherit the sample time from the input, if there is one,

or the Simulink software model (when there are no input ports on the block). Enter the sample time in seconds as you need.

Memory Copy

Options Parameters



Function Block Parameters: Memory Copy

Memory Copy (mask)

Write/read to/from sequential locations of the target memory starting at specified start address and offset using specified data length and stride. The start address and offset can be changed during run-time. Memory may be written/read during initialization, termination and at every sample time. You can specify custom C source code to be inserted before and/or after the memory write/read instruction(s). Quick DMA (QDMA) data copy can be used on supported DSP platforms.

Source | Destination | Options

- Set memory value at initialization
Specify initialization value source: Specify constant value
Initialization value (constant): 1
- Apply initialization value as mask
Bitwise operator: bitwise AND
- Set memory value at termination
Termination value: 1
- Set memory value only at initialization/termination
- Insert custom code before memory write
Custom code: /* Custom Code Before Write*/
- Insert custom code after memory write
Custom code: /* Custom Code After Write*/
- Use QDMA for copy (if available)
- Enable blocking mode

OK Cancel Help Apply

Set memory value at initialization

When you check this option, you direct the block to initialize the memory location to a specific value when you initialize your program at run time. After you select this option, use the **Set memory value at termination** and **Specify initialization value source** parameters to set your desired value. Alternately, you can tell the block to get the initial value from the block input.

Specify initialization value source

After you check Set memory value at initialization, use this parameter to select the source of the initial value. Choose either

- **Specify constant value** — Sets a single value to use when your program initializes memory. Enter any value that meets your needs.
- **Specify source code symbol** — Specifies a variable (a symbol) to use for the initial value. Enter the symbol as a string.

Initialization value (constant)

If you check **Set memory value at initialization** and choose **Specify constant value** for **Specify initialization value source**, enter the constant value to use in this field. Any real value that meets your needs is acceptable.

Initialization value (source code symbol)

If you check **Set memory value at initialization** and choose **Specify source code symbol** for **Specify initialization value source**, enter the symbol to use in this field. Any symbol that meets your needs and is in the symbol table for the program is acceptable. When you enter the symbol, the block does not verify whether the symbol is a valid one. If it is not valid you get an error when you try to compile, link, and run your generated code.

Apply initialization value as mask

You can use the initialization value as a mask to manipulate register contents at the bit level. Your initialization value is treated as a string of bits for the mask.

Memory Copy

Checking this parameter enables the **Bitwise operator** parameter for you to define how to apply the mask value.

To use your initialization value as a mask, the output from the copy has to be a specific address. It cannot be an output port, but it can be a symbol.

Bitwise operator

To use the initialization value as a mask, select one of the entries on the following table from the **Bitwise operator** list to describe how to apply the value as a mask to the memory value.

Bitwise Operator List Entry	Description
bitwise AND	Apply the mask value as a bitwise AND to the value in the register.
bitwise OR	Apply the mask value as a bitwise OR to the value in the register.
bitwise exclusive OR	Apply the mask value as a bitwise exclusive OR to the value in the register.
left shift	Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer.
right shift	Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer.

Applying a mask to the copy process lets you select individual bits in the result, for example, to read the value of the fifth bit by applying the mask.

Set memory value at termination

Along with initializing memory when the program starts to access this memory location, this parameter directs the program to set memory to a specific value when the program terminates.

Set memory value only at initialization/termination

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. When you check this option, the block only does the memory initialization and termination processes. It does not perform any copies during real-time operations.

Insert custom code before memory write

Select this parameter to add custom ANSI C code before the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

Custom code

Enter the custom ANSI C code to insert into the generated code just before the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

Insert custom code after memory write

Select this parameter to add custom ANSI C code immediately after the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

Custom code

Enter the custom ANSI C code to insert into the generated code just after the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

Memory Copy

Use QDMA for copy (if available)

For processors that support quick direct memory access (QDMA), select this parameter to enable the QDMA operation and to access the blocking mode parameter.

If you select this parameter, your source and destination data types must be the same or the copy operation returns an error. Also, the input and output stride values must be one.

Enable blocking mode

If you select the **Use QDMA for copy** parameter, select this option to make the memory copy operations blocking processes. With blocking enabled, other processing in the program waits while the memory copy operation finishes.

See Also

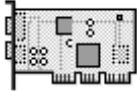
Memory Allocate

Purpose

Configure model for a supported processor

Library

Description



Custom Board

Use this block to configure hardware settings and code generation features for your custom board. Include this block in models you use to generate Real-Time Workshop code to run on processors and boards. It does not connect to any other blocks, but stands alone to set the processor preferences for the model.

Note Simulink and Embedded IDE Link software return an error when your model does not include a Target Preferences block or has more than one. When you are generating code for a model, place the Target Preferences block at the top level of your model. When you are generating code for a subsystem, place the Target Preferences block at the subsystem level of your model.

The processor options you specify on this block are:

- Processor and board information
- Memory mapping and layout

Setting the options included in this dialog box results in identifying your processor and board to Real-Time Workshop software, Embedded IDE Link, and Simulink software. Setting the options also, configures the memory map for your processor. Both steps are essential for generating code for any board that is custom or explicitly supported.

Generating Code from Model Subsystems

Real-Time Workshop software provides the ability to generate code from a selected subsystem in a model. To generate code for custom hardware from a subsystem, the subsystem model must include a Target Preferences block.

Target Preferences/Custom Board

Dialog Box

This reference page section contains the following subsections:

- “Board Pane” on page 9-46
- “Memory Pane” on page 9-49
- “Sections Pane” on page 9-52
- “Add Processor Dialog Box” on page 9-54

Target Preferences block dialog boxes provide tabbed access to the following panes with options you set for the processor and board:

- Board Pane — Select the processor, set the clock speed, and identify the processor. In addition, **Add new** on this pane opens the New Processor dialog box.
- Memory Pane — Set the memory allocation and layout on the processor (memory mapping).
- Sections Pane — Determine the arrangement and location of the sections on the processor and compiler information.

Board Pane

The following options appear on the **Board** pane, under the **Board Properties**, **Board Support**, and **IDE Support** labels.

Board type

Enter the type of your target board. Enter Custom to support any board that uses a processor on the **Processor** list, or enter the name of a supported board. If you are using one of the explicitly supported boards, choose the Target Preferences/Custom Board block for that board from the Simulink .

Processor

Select the type of processor to use from the list. The processor you select determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box.

Add New

Clicking **Add new** opens a new dialog box where you specify configuration information for a processor that is not on the Processor list.

For details about the New Processor dialog box, refer to “Add Processor Dialog Box” on page 9-54.

Delete

Delete a processor that you added to the **Processor** list. You cannot delete processors that you did not add.

CPU Clock (MHz)

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting the actual clock rate produces code that runs correctly on the hardware. Setting this value incorrectly causes timing and profiling errors when you run the code on the hardware.

The timer uses the value of **CPU clock** to calculate the time for each interrupt. For example, a model with a sine wave generator block running at 1 kHz uses timer interrupts to generate sine wave samples at the proper rate. For example, using 100 MHz, the timer calculates the sine generator interrupt period as follows:

- Sine block rate = 1 kHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires:

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 100,000 \text{ clock ticks}$$

Thus, report the correct clock rate, or the interrupts come at the wrong times and the results are incorrect.

Target Preferences/Custom Board

Board Support

Select the following parameters and edit their values in the text box on the right:

- **Source files** — Enter the full paths to source code files.
- **Include paths** — Add paths to include files.
- **Libraries** — Identify specific libraries for the processor. Required libraries appear on the list by default. To add more libraries, entering the full path to the library with the library file in the text area.
- **Initialize functions** — If your project requires an initialize function, enter it in this field. By default, this parameter is empty.
- **Terminate functions** — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

Note Invalid or incorrect entries in these fields can cause errors during code generation. When you enter a file path, library, or function, the block does not verify that the path or function exists or is valid.

When entering a path to a file, library, or other custom code, use the following string in the path to refer to the CCS installation directory.

```
$(install_dir)
```

Enter new paths or files (custom code items) one entry per line. Include the full path to the file for libraries and source code.

Board custom code options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

Operating System

The software disables this option if a supported RTOS is not available for your processor.

Board name

Board name appears after you click **Get from IDE**. Select the board you are using. Match **Board name** with the **Board Type** option near the top of the **Board** pane.

Processor name

Processor name appears after you click **Get from IDE**. If the board you selected in **Board name** has multiple processors, select the processor you are using. Match **Processor name** with the **Processor** option near the top of the **Board** pane.

Note Click **Apply** to update the board and processor description under **IDE Support**.

Memory Pane

After selecting a board, specify the layout of the physical memory on your processor and board to determine how to use it for your program. For supported boards, the board-specific Target Preferences blocks set the default memory map.

The **Memory** pane contains memory options for:

- **Physical Memory** — Specifies the processor and board memory map
- **Cache Configuration** — Select a cache configuration where available, such as L2 cache, and select one of the corresponding configuration options, such as 32 kb.

For more information about memory segments and memory allocation, consult the reference documentation for the IDE or processor.

The **Physical Memory** table shows the memory segments (or “memory banks”) available on the board and processor. By default, Target

Target Preferences/Custom Board

Preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured Target Preferences blocks shows the memory segments available on the board, but external to the processor. Target Preferences blocks set default starting addresses, lengths, and contents of the default memory segments.

Click **Add** to add physical memory segments to the **Memory banks** table.

After you add the segment, you can configure the starting address, length, and contents for the new segment.

Name

To change the memory segment name, click the name and type the new name. Names are case sensitive. `NewSegment` is not the same as `newsegment` or `newSegment`.

Note You cannot rename default processor memory segments (name in gray text).

Address

Address reports the starting address for the memory segment showing in **Name**. Address entries are in hexadecimal format and limited only by the board or processor memory.

Length

From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs).

Contents

Configure the segment to store Code, Data, or Code & Data. Changing processors changes the options for each segment.

You can add and use as many segments of each type as you need, within the limits of the memory on your processor. Every processor must have a segment that holds code, and a segment that holds data.

Add

Click **Add** to add a new memory segment to the processor memory map. When you click **Add**, a new segment name appears, for example NEWMEM1, in **Name** and on the **Memory banks** table. In **Name**, change the temporary name NEWMEM1 by entering the new segment name. Entering the new name, or clicking **Apply**, updates the temporary name on the table to the name you enter.

Remove

This option lets you remove a memory segment from the memory map. Select the segment to remove on the **Memory banks** table and click **Remove** to delete the segment.

Cache (Configuration)

When the **Processor** on the Board pane supports an L2 cache memory structure, the dialog box displays a table of **Cache** parameters. You can use this table to configure the cache as SRAM and partial cache. Both the data memory and the program share this second-level memory.

If your processor supports the two-level memory scheme, this option enables the L2 cache on the processor.

Some processors support code base memory organization. For example, you can configure part of internal memory as code.

Cache level lets you select one of the available cache levels to configure by selecting one of its configurations. For example, you can select L2 cache level and choose one of its configurations, such as 32 kb.

Target Preferences/Custom Board

Sections Pane

Options on this pane specify where program sections go in memory. Program sections are distinct from memory segments—sections are portions of the executable code stored in contiguous memory locations. Commonly used sections include `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler and some can be custom sections.

For more information about program sections and objects, refer to the online help for your IDE.

Within the Sections pane, you configure the allocation of sections for **Compiler** and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections** and **Custom sections** lists in the pane. All sections do not appear on all lists.

String	Section List	Description of the Section Contents
<code>.bss</code>	Compiler	Static and global C variables in the code
<code>.cinit</code>	Compiler	Tables for initializing global and static variables and constants
<code>.cio</code>	Compiler	Standard I/O buffer for C programs
<code>.const</code>	Compiler	Data defined with the C qualifier and string constants
<code>.data</code>	Compiler	Program data for execution
<code>.far</code>	Compiler	Variables, both static and global, defined as far variables
<code>.pinit</code>	Compiler	Load allocation of the table of global object constructors section
<code>.stack</code>	Compiler	The global stack
<code>.switch</code>	Compiler	Jump tables for switch statements in the executable code

Target Preferences/Custom Board

String	Section List	Description of the Section Contents
.system	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants

You can learn more about memory sections and objects in the online help for your IDE.

Default Sections

When you highlight a section on the list, **Description** show a brief description of the section. Also, **Placement** shows you the memory allocation of the section.

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

Shows the allocation of the selected **Compiler sections** entry in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments as defined in the physical memory map on the **Memory** pane. Select one of the listed memory segments to allocate the highlighted compiler section to the segment.

To see a description of the placement item, hover your mouse pointer over the item for a few moments.

Custom Sections

If your program uses code or data sections that are not in the **Compiler sections**, add the new sections to **Custom sections**.

Sections

This window lists data sections that are not in the **Compiler sections**.

Target Preferences/Custom Board

Placement

With your new section added to the **Name** list, select the memory segment to which to add your new section. Within the restrictions imposed by the hardware and compiler, you can select any segment that appears on the list.

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. You can also click **OK** to add the section to the list and close the dialog box.

Name

Enter the name of the new section here. To add a new section, click **Add**. Then, replace the temporary name with the name to use. Although the temporary name includes a period at the beginning you do not need to include the period in your new name. Names are case sensitive. `NewSection` is not the same as `newsection`, or `newSection`.

Contents

Identify whether the contents of the new section are **Code**, **Data**, or **Any**.

Remove

To remove a section from the **Custom sections** list, select the section and click **Remove**.

Add Processor Dialog Box

To add a new processor to the drop down list for the **Processors** option, click the **Add new** button on the **Board** pane. The software opens the **Add Processor** dialog box.

New Name

Provide a name to identify your new processor. You can use any valid C string value in this field. The name you enter in this field appears on the list of processors after you add the new processor.

If you do not provide an entry for each parameter, Embedded IDE Link returns an error message without creating a processor entry.

Based On

When you add a processor, the dialog box uses the settings from the currently selected processor as the basis for the new one. This parameter displays the currently selected processor.

Compiler options

Identifies the processor family of the new processor to the compiler. Successful compilation requires this switch. The string depends on the processor family or class.

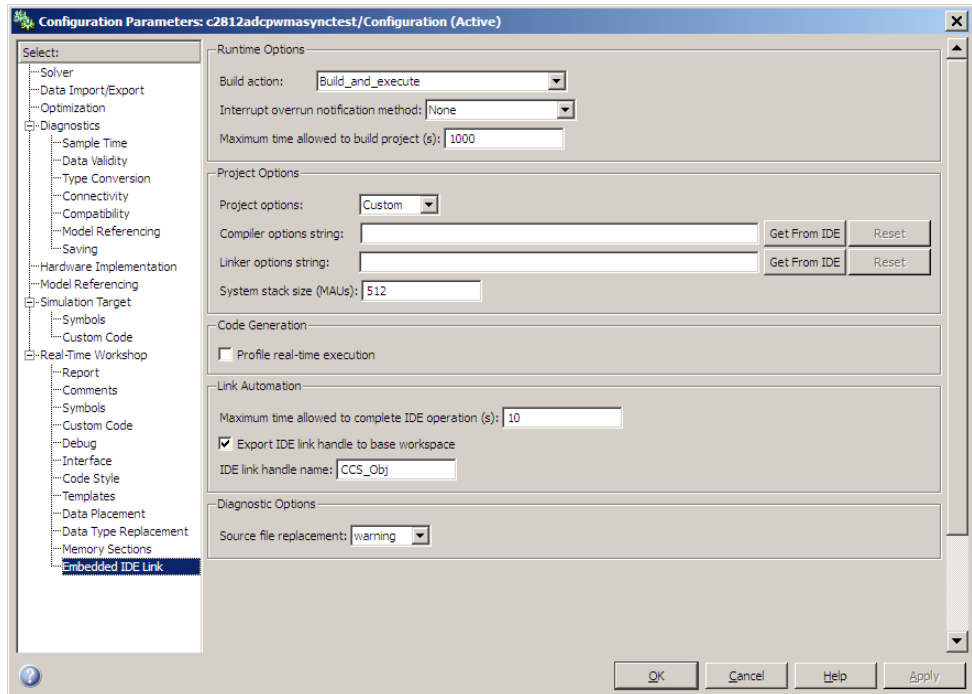
Linker options

Identifies the processor family of the new processor to the compiler. Successful compilation requires this switch. The string depends on the processor family or class.

Target Preferences/Custom Board

Configuration Parameters

Embedded IDE Link Pane



In this section...

- “Overview” on page 10-4
- “Export IDE link handle to base workspace” on page 10-5
- “IDE link handle name” on page 10-7
- “Profile real-time execution” on page 10-8
- “Profile by” on page 10-10
- “Number of profiling samples to collect” on page 10-12
- “Project options” on page 10-14
- “Compiler options string” on page 10-16
- “Linker options string” on page 10-18

In this section...

“System stack size (MAUs)” on page 10-20

“Build action” on page 10-21

“Interrupt overrun notification method” on page 10-24

“Interrupt overrun notification function” on page 10-26

“PIL block action” on page 10-27

“Maximum time allowed to build project (s)” on page 10-29

“Maximum time to complete IDE operations (s)” on page 10-31

“Source file replacement” on page 10-33

Overview

Options on this pane configure the generated projects and code for the processors supported by your embedded IDE. They also enable PIL block generation and provide real-time execution and stack use profiling.

Export IDE link handle to base workspace

Directs the software to export the `ticcs` object to your MATLAB workspace.

Settings

Default: On



Directs the build process to export the `ticcs` object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.



prevents the build process from exporting the `ticcs` object to your MATLAB software workspace.

Dependency

This parameter enables **IDE link handle name**.

Command-Line Information

Parameter: `exportIDEObj`

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

IDE link handle name

specifies the name of the ticcs object that the build process creates.

Settings

Default: CCS_Obj

- Enter any valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the ticcs object.
- The handle name is case sensitive.

Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

Command-Line Information

Parameter: ideObjName

Type: string

Value:

Default: CCS_Obj

Recommended Settings

Application	Setting
Debugging	Enter any valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Profile real-time execution

enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

Settings

Default: Off



On

Adds instrumentation to the generated code to support execution profiling and generate the profiling report.



Off

Does not instrument the generated code to produce the profile report.

Dependencies

This parameter adds **Number of profiling samples to collect** and **Profile by**.

Selecting this parameter disables **Export ID link handle to base workspace**.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop` project removes this parameter.

Command-Line Information

Parameter: ProfileGenCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics in the Embedded IDE Link User’s Guide..

Profile by

Defines which execution profiling technique to use.

Settings

Default: Task

Task

Profiles model execution by the tasks in the model.

Atomic subsystem

Profiles model execution by the atomic subsystems in the model.

Dependencies

Selecting **Real-time execution profiling** enables this parameter.

Command-Line Information

Parameter: profileBy

Type: string

Value: Task | Atomic subsystem

Default: Task

Recommended Settings

Application	Setting
Debugging	Task or Atomic subsystem
Traceability	Archive_library
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic in the Embedded IDE Link User’s Guide.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics in the Embedded IDE Link User’s Guide..

Number of profiling samples to collect

Specifies the number of profiling samples to collect. Collection stops when the buffer for profiling data is full.

Settings

Default: 100

Minimum: 1

Maximum: Buffer capacity in samples

Tips

- Collecting profiling data on a simulator may take a very long time.
- Data collection stops after collecting the specified number of samples. The application and processor continue to run.

Dependencies

This parameter is enabled by **Profile real-time execution**.

Command-Line Information

Parameter: ProfileNumSamples

Type: int

Value: Positive integer

Default: 100

Recommended Settings

Application	Setting
Debugging	100
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Project options

Sets the project options for building your project from the model.

Settings

Default: Custom

Custom

Lets the user apply a specialized combination of build and optimization settings.

Custom applies the same settings as the Release project configuration in CCS, except:

- The compiler options do not use any optimizations.
- The memory configuration specifies a memory model that uses Far Aggregate for data and Far for functions.

Debug

Applies the Debug project options defined by Code Composer Studio software to the generated project and code. The Compiler options string becomes `-g -d _DEBUG`

Release

Applies the Release project configuration defined by Code Composer Studio software to the generated project and code. Sets the **Compiler options** string to `-o2`.

Dependencies

- Selecting Custom disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting Release sets the **Compiler options string** to `-o2`.
- Selecting Debug sets the **Compiler options string** to `-g -d _DEBUG`

Command-Line Information

Parameter: projectOptions

Type: string
Value: Custom | Debug | Release
Default: Custom

Recommended Settings

Application	Setting
Debugging	Custom or Debug
Traceability	Custom, Debug, Release
Efficiency	Release
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Compiler options string

Lets you enter a string of compiler options to define your project configuration.

Settings

Default: No default

Tips

- To import compiler string options from the current project in CCS, click **Get from IDE**.
- To reset the compiler options to the default values, click **Reset**.
- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Project options** to **Custom** applies the **Custom** compiler options defined by Embedded IDE Link software. **Custom** does not use any optimizations.
- Setting **Project options** to **Debug** applies the `_Debug`, `-g`, and `-d` compiler flags defined by Code Composer Studio software.
- Setting **Project options** to **Release** applies the CCS Release compiler options and adds the `-o2` optimization flag defined by Code Composer Studio software.

Command-Line Information

Parameter: compilerOptionsStr

Type: string

Value: Custom | Debug | Release

Default: Custom

Recommended Settings

Application	Setting
Debugging	Custom

Application	Setting
Traceability	Custom
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Linker options string

Enables you to specify linker command options that determine how to link your project when you build your project.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.
- To import linker string options from the current project in CCS, click **Get from IDE**.
- To reset the linker command options to the default values, click **Reset**.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information

Parameter: linkerOptionsStr

Type: string

Value: any valid linker option

Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

System stack size (MAUs)

Allocates memory for the system stack on the processor.

Settings

Default: 8192

Minimum: 0

Maximum: Available memory

- Enter the stack size in minimum addressable units (MAUs)..
- The software does not verify that your size is valid. Be sure that you enter an acceptable value.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information

Parameter: `systemStackSize`

Type: `int`

Default: 8192

Recommended Settings

Application	Setting
Debugging	<code>int</code>
Traceability	<code>int</code>
Efficiency	<code>int</code>
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Build action

Defines how Real-Time Workshop software responds when you press Ctrl+B to build your model.

Settings

Default: Build_and_execute

Build_and_execute

Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

Create_project

Directs Real-Time Workshop software to create a new project in the IDE.

Archive_library

Invokes the CCS Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

Build

Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

Create_processor_in_the_loop_project

Directs the Real-Time Workshop code generation process to create PIL algorithm object code as part of the project build.

Dependencies

Selecting Archive_library removes the following parameters:

- **Interrupt overrun notification method**
- **Interrupt overrun notification function**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**

- **Reset**
- **Export IDE link handle to base workspace**

Selecting `Create_processor_in_the_loop_project` removes the following parameters:

- **Interrupt overrun notification method**
- **Interrupt overrun notification function**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

Command-Line Information

Parameter: `buildAction`

Type: `string`

Value: `Build` | `Build_and_execute` | `Create_project` | `Archive_library`
| `Create_processor_in_the_loop_project`

Default: `Build_and_execute`

Recommended Settings

Application	Setting
Debugging	<code>Build_and_execute</code>
Traceability	<code>Archive_library</code>
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic in the Embedded IDE Link User’s Guide.

Interrupt overrun notification method

Specifies how your program responds to overrun conditions during execution.

Settings

Default: None

None

Your program does not notify you when it encounters an overrun condition.

Print_message

Your program prints a message to standard output when it encounters an overrun condition.

Call_custom_function

When your program encounters an overrun condition, it executes a function that you specify in **Interrupt overrun notification function**.

Tips

- The definition of the standard output depends on your configuration.

Dependencies

Selecting `Call_custom_function` enables the **Interrupt overrun notification function** parameter.

Setting this parameter to `Call_custom_function` enables the **Interrupt overrun notification function** parameter.

Command-Line Information

Parameter: `overrunNotificationMethod`

Type: `string`

Value: `None | Print_message | Call_custom_function`

Default: `None`

Recommended Settings

Application	Setting
Debugging	Print_message or Call_custom_function
Traceability	Print_message
Efficiency	None
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Interrupt overrun notification function

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

Settings

No Default

Dependencies

This parameter is enabled by setting **Interrupt overrun notification method** to `Call_custom_function`.

Command-Line Information

Parameter: `overrunNotificationFcn`

Type: string

Value: no default

Default: no default

Recommended Settings

Application	Setting
Debugging	String
Traceability	String
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

PIL block action

Specifies whether Real-Time Workshop software builds the PIL block and downloads the block to the processor

Settings

Default: Create_PIL_block_and_download

Create_PIL_block_build_and_download

Builds and downloads the PIL application to the processor after creating the PIL block. Adds PIL interface code that exchanges data with Simulink.

Create_PIL_block

Creates a PIL block, places the block in a new model, and then stops without building or downloading the block. The resulting project will not compile in the IDE.

None

Configures model to generate a CCS project that contains the PIL algorithm code. Does not build the PIL object code or block. The new project will not compile in the IDE.

Tips

- When you click **Build** on the PIL dialog box, the build process adds the PIL interface code to the project and compiles the project in the IDE.
- If you select **Create PIL block**, you can build manually from the block right-click context menu
- After you select **Create PIL Block**, *copy* the PIL block into your model to replace the original subsystem. Save the original subsystem in a different model so you can restore it in the future. Click **Build** to build your model with the PIL block in place.
- *Add* the PIL block to your model to use cosimulation to compare PIL results with the original subsystem results. Refer to the demo “Comparing Simulation and processor Implementation with Processor-in-the-Loop (PIL)” in the product demos Embedded IDE Link

- When you select `None` or `Create_PIL_block`, the generated project will not compile in the IDE. To use the PIL block in this project, click **Build** followed by **Download** in the PIL block dialog box.

Dependency

Enable this parameter by setting **Build action** to `Create_processor_in_the_loop_project`.

Command-Line Information

Parameter: `configPILBlockAction`

Type: `string`

Value: `None` | `Create_PIL_block` | `Create_PIL_block_build_and_download`

Default: `Create_PIL_block`

Recommended Settings

Application	Setting
Debugging	<code>Create_PIL_block_build_and_download</code>
Traceability	<code>Create_PIL_block_build_and_download</code>
Efficiency	<code>None</code>
Safety precaution	No impact

See Also

For more information, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic in the Embedded IDE Link User’s Guide.

Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

Settings

Default: 1000

Minimum: 1

Maximum: No limit

Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a ticcs object or the **Maximum time to complete IDE operations** timeout value.

Dependency

This parameter is disabled when you set **Build action** to Create_project.

Command-Line Information

Parameter:TBD

Type: int

Value: Integer greater than 0

Default: 100

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Maximum time to complete IDE operations (s)

specifies how long the software waits for IDE functions, such as read or write, to return completion messages.

Settings

Default: 10

Minimum: 1

Maximum: No limit

Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `ticcs` object or the **Maximum time allowed to build project (s)** timeout value

Command-Line Information

Parameter:TBD

Type: int

Value:

Default: 10

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Source file replacement

Selects the diagnostic action to take if Embedded IDE Link software detects conflicts that you are replacing source code with custom code.

Settings

Default: warn

none

Does not generate warnings or errors when it finds conflicts.

warning

Displays a warning.

error

Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

Tips

- The build operation continues if you select `warning` and the software detects custom code replacement. You see warning messages as the build progresses.
- Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select `none` when the replacement process is correct and you do not want to see multiple messages during your build.
- The messages apply to Real-Time Workshop **Custom Code** replacement options as well.

Command-Line Information

Parameter: DiagnosticActions

Type: string

Value: none | warning | error

Default: warning

Recommended Settings

Application	Setting
Debugging	error
Traceability	error
Efficiency	warning
Safety precaution	error

See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

Supported Processors

This appendix provides the details about the processors, simulators, and software that work with Embedded IDE Link.

- “Supported Platforms” on page A-2
- “Supported Versions of Code Composer Studio” on page A-5

Supported Platforms

In this section...
“Product Features Supported by Each Processor or Family” on page A-2
“Coemulation Support” on page A-3
“Supported Processors and Simulators” on page A-3
“Custom Board Support” on page A-4

This appendix lists the processors and simulators that work with the latest released version of Embedded IDE Link. Generally, the product supports boards and simulators from a given processor family. In some cases, only the simulators work, as noted in the tables in the next sections.

Product Features Supported by Each Processor or Family

The following table indicates which Embedded IDE Link features are available by processor family.

Features by Processor Family

Automation Interface Component			Project Generator Component	Verification	
Processor Family	Debug Mode	RTDX	Code Generation	Processor-in-the-Loop	Real-Time Execution Profiling
C28xx	Yes	Yes	Yes	Yes	Yes
C54xx	Yes	No	No	No	No
C55xx	Yes	Yes	Yes	Yes	Yes
C62xx	Yes	No	Yes	Yes	Yes
C64x and C64x+	Yes	No	Yes	Yes	Yes

Features by Processor Family (Continued)

Automation Interface Component			Project Generator Component	Verification	
Processor Family	Debug Mode	RTDX	Code Generation	Processor-in-the-Loop	Real-Time Execution Profiling
C67x and C67x+	Yes	No	Yes	Yes	Yes
DM64x	Yes	No	Yes	Yes	Yes
DM643x	Yes	No	Yes	Yes	Yes
TMS470R1x	Yes	No	No	No	No
TMS470R2x	Yes	No	No	No	No

Coemulation Support

An added feature for OMAP processors is coemulation for the two processors that comprise the OMAP. Embedded IDE Link supports coemulation or direct multiprocessor support for the TMS470R2x (TI-enhanced ARM925) and TMS320C55x DSP in OMAP 1510 and OMAP 5910.

Supported Processors and Simulators

Embedded IDE Link for has been tested on the following processors and boards produced by TI and others.

- TMS320C2000
 - Simulators (C28x)
 - C2808 eZdsp, C2812 eZdsp, C2833x Floating-Point Processors
- TMS320C5000
 - Simulators (C54x, C55x)
 - C5510 DSK
 - C5416 DSK, C5402 DSK

- TMS320C6000
 - Simulators (C62x, C64x, C67x)
 - C6713 DSK, C6711 DSK, C6701 EVM
 - C6416 DSK
 - DM64x
 - DM643x
 - C6211 DSK
- OMAP
 - OMAP 1510
 - OMAP 5910
- TMS470Rxx
 - Boards and simulators based on the TMS470R1x processor
 - Boards and simulators based on the TMS470R2x processor

Custom Board Support

You can use Embedded IDE Link with your custom board if:

- It uses one or more of the supported processors in the preceding list or if it is in the **Processor** list of the Target Preferences/Custom Board block for your processor family.
- You are able to use Code Composer Studio IDE to interact with your board/processor combination.

you should be able to use Embedded IDE Link with your hardware.

Supported Versions of Code Composer Studio

The following table lists versions of Embedded IDE Link and the versions of Code Composer Studio they support.

Embedded IDE Link Version	MATLAB Release	Supported Version of Code Composer Studio
4.0	R2009b	CCS 3.3 for C64x+, C6000, C5000, C2000, OMAP processors (tested on CCS 3.3 SR10)
3.4	R2009a	CCS 3.3 for C64x+, C6000, C5000, C2000, OMAP processors
3.3	R2008b	Only CCS 3.3 with DSP/BIOS 5.32.01 or 5.32.05 (not 5.32.00) (C64x+, C6000, C5000, C2000, OMAP) CCS 3.3 SR7 has a bug and is not supported
3.2	R2008a	Only CCS 3.3 with DSP/BIOS 5.3 (not 5.32.00)
3.1	R2007b	Only CCS 3.3 with DSP/BIOS 5.3
3.0	R2007a	<ul style="list-style-type: none"> • CCS 3.2 for C64x+ processors • CCS 3.1 for C2000, C5000, C6000, and OMAP processors
2.1	R2006b	<ul style="list-style-type: none"> • CCS 3.2 for C64x+ processors • CCS 3.1 for C2000, C5000, C6000, and OMAP processors
2.0	R2006a+	CCS 3.1 for C2000, C5000, C6000, and OMAP processors
1.5	R2006a	CCS 3.1 for C2000, C5000, C6000, and OMAP processors
1.4.2	R14SP3	<ul style="list-style-type: none"> • CCS 3.0 for C6000 processors • CCS 2.2 for C2000, C5000, C6000, and OMAP processors

Embedded IDE Link Version	MATLAB Release	Supported Version of Code Composer Studio
1.4.1	R14SP2	<ul style="list-style-type: none">• CCS 3.0 for C6000 processors• CCS 2.2 for C2000, C5000, C6000, and OMAP processors
1.4	R14SP1+	<ul style="list-style-type: none">• CCS 3.0 for C6000 processors• CCS 2.2 for C2000, C5000, C6000, and OMAP processors
1.3.2	R14SP1	<ul style="list-style-type: none">• CCS 2.2 for C2000, C5000, C6000, and OMAP processors• CCS 2.12 for C2000, C5000, C6000, and OMAP processors
1.3.1	R14	<ul style="list-style-type: none">• CCS 2.2 for C2000, C5000, C6000, and OMAP processors• CCS 2.12 for C2000, C5000, C6000, and OMAP processors
1.3	R13SP1+	CCS 2.12 for C2000, C5000, C6000, and OMAP processors

Reported Limitations and Tips

Reported Issues

In this section...
“Demonstration Programs Do Not Run Properly Without Correct GEL Files” on page B-3
“Error Accessing type Property of ticcs Object Having Size Greater Than 1” on page B-3
“Changing Values of Local Variables Does Not Take Effect” on page B-4
“Code Composer Studio Cannot Find a File After You Halt a Program” on page B-4
“C54x XPC Register Can Be Modified Only Through the PC Register” on page B-6
“Working with More Than One Installed Version of Code Composer Studio” on page B-6
“Changing CCS Versions During a MATLAB Session” on page B-7
“MATLAB Hangs When Code Composer Studio Cannot Find a Board” on page B-7
“Using Mapped Drives” on page B-9
“Uninstalling Code Composer Studio 3.3 Prevents Embedded IDE Link From Connecting” on page B-9

Some long-standing issues affect the Embedded IDE Link product. When you are using `ticcs` objects and the software methods to work with Code Composer Studio and supported hardware or simulators, recall the information in this section.

The latest issues in the list appear at the bottom. HIL refers to “hardware in the loop,” also called processor in the loop (PIL) here and in other applications, and sometimes referred to as function calls.

Demonstration Programs Do Not Run Properly Without Correct GEL Files

To run the Embedded IDE Link demos, you must load the appropriate GEL files before you run the demos. For some boards, the demos run fine with the default CCS GEL file. Some boards need to run device-specific GEL files for the demos to work correctly.

Here are demos and boards which require specific GEL files.

- Board: C5416 DSK
Demos: `rtdxtutorial`, `rtdx1msdemo`
Emulator: XDS-510
GEL file to load: `c5416_dsk.gel`

In general, if a demo does not run correctly with the default GEL file, try using a device-specific GEL file by defining the file in the CCS Setup Utility.

Error Accessing type Property of ticcs Object Having Size Greater Than 1

When `cc` is a `ticcs` object consisting of an array of single `ticcs` objects such that

```
cc
Array of TICCS Objects:
  API version : 1.2
  Board name  : C54x Simulator (Texas Instruments)
  Board number : 0
  Processor 0 (element 1) : TMS320C5407 (CPU, Not Running)
  Processor 0 (element 2) : TMS320C5407 (CPU, Not Running)
```

you cannot use `cc` to access the type object. The example syntaxes below generate errors.

- `cc.type`
- `add(cc.type, 'mytypedef', 'int')`

To access `type` without the error, reference the individual elements of `cc` as follows:

- `cc(1).type`
- `add(cc(2).type, 'mytypedef', 'int')`

Changing Values of Local Variables Does Not Take Effect

If you halt the execution of your program on your DSP and modify a local variable's value, the new value may not be acknowledged by the compiler. If you continue to run your program, the compiler uses the original value of the variable.

This problem happens only with local variables. When you write to the local variable via the Code Composer Studio Watch Window or via a MATLAB object, you are writing into the variable's absolute location (register or address in memory).

However, within the processor function, the compiler sometimes saves the local variable's values in an intermediate location, such as in another register or to the stack. That intermediate location cannot be determined or changed/updated with a new value during execution. Thus the compiler uses the old, unchanged variable value from the intermediate location.

Code Composer Studio Cannot Find a File After You Halt a Program

When you halt a running program on your processor, Code Composer Studio may display a dialog box that says it cannot find a source code file or a library file.

When you halt a program, CCS tries to display the source code associated with the current program counter. If the program stops in a system library like the runtime library, DSP/BIOS, or the board support library, it cannot find the source code for debug. You can either find the source code to debug it or select the **Don't show this message again** checkbox to ignore messages like this in the future.

For more information about how CCS responds to the halt, refer the online Help for CCS. In the online help system, use the search engine to search for the keywords “Troubleshooting” and “Support.” The following information comes from the online help for CCS, starting with the error message:

File Not Found

The debugger is unable to locate the source file necessary to enable source-level debugging for this program.

To specify the location of the source file

- 1** Click **Yes**. The Open dialog box appears.
- 2** In the Open dialog box, specify the location and name of the source file then click **Open**.

The next section provides more details about file paths.

Defining a Search Path for Source Files

The Directories dialog box enables you to specify the search path the debugger uses to find the source files included in a project.

To Specify Search Path Directories

- 1** Select **Option > Customize**.
- 2** In the Customize dialog box, select the **Directories** tab. Use the scroll arrows at the top of the dialog box to locate the tab.

The Directories dialog box offers the following options.

- **Directories.** The **Directories** list displays the defined search path. The debugger searches the listed directories in order from top to bottom.

If two files have the same name and are located in different directories, the file located in the directory that appears highest in the **Directories** list takes precedence.

- **New.** To add a new directory to the **Directories** list, click **New**. Enter the full path or click **browse [...]** to navigate to the appropriate directory. By default, the new directory is added to the bottom of the list.
- **Delete.** Select a directory in the **Directories** list, then click **Delete** to remove that directory from the list.
- **Up.** Select a directory in the **Directories** list, then click **Up** to move that directory higher in the list.
- **Down.** Select a directory in the **Directories** list, then click **Down** to move that directory lower in the list.

3 Click **OK** to close the **Customize** dialog box and save your changes.

C54x XPC Register Can Be Modified Only Through the PC Register

You cannot modify the XPC register value directly using `regwrite` to write into the register. When you are using extended program addressing in C54x, you can modify the XPC register by using `regwrite` to write a 23-bit data value in the PC register. Along with the 16-bit PC register, this operation also modifies the 7-bit XPC register that is used for extended program addressing. On the C54x, the PC register is 23 bits (7 bits in XPC + 16 bits in PC).

You can then read the XPC register value using `regread`.

Working with More Than One Installed Version of Code Composer Studio

When you have more than one version of Code Composer Studio installed on your machine, you cannot select which CCS version MATLAB Embedded IDE Link attaches to when you create a `ticcs` object. If, for example, you have both CCS for C5000 and CCS for C6000 versions installed, you cannot choose to connect to the C6000 version rather than the C5000 version.

When you issue the command

```
cc = ticcs
```

Embedded IDE Link starts the CCS version you last used. If you last used your C5000 version, the `cc` object accesses the C5000 version.

Workaround

To make your `ticcs` object access the correct processor:

- 1 Start and close the appropriate CCS version before you create the `ticcs` object in MATLAB.
- 2 Create the `ticcs` object using the `boardnum` and `procnum` properties to select your processor, if needed.

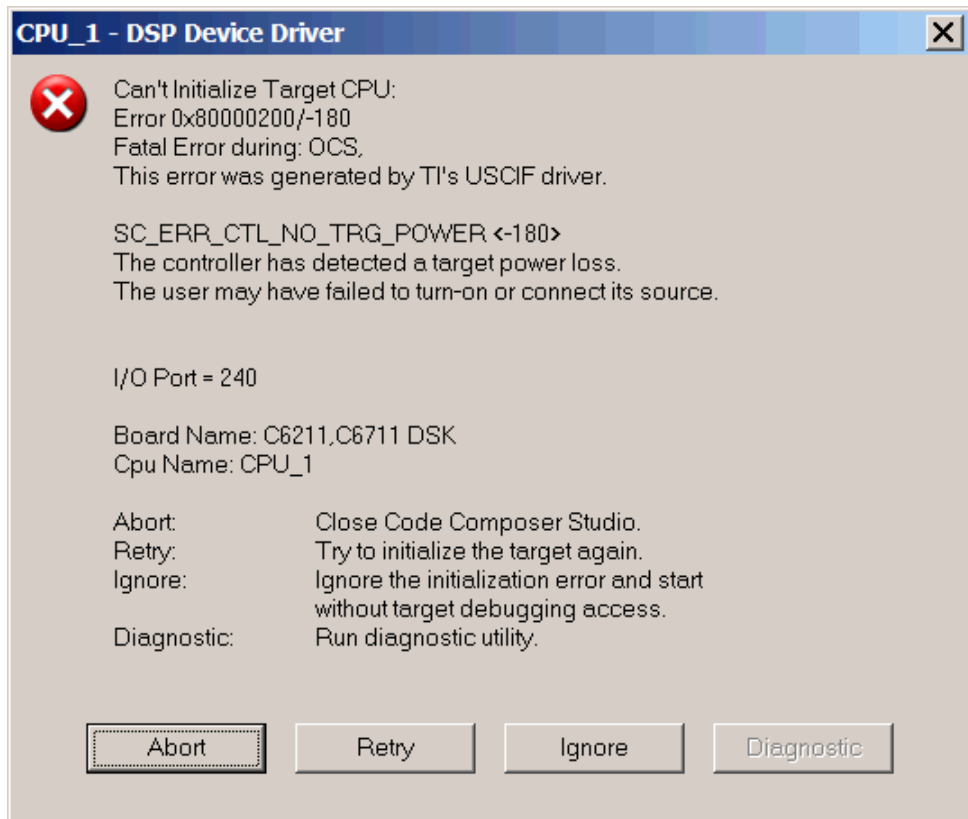
Recall that `ccsboardinfo` returns the `boardnum` and `procnum` values for the processors that CCS recognizes.

Changing CCS Versions During a MATLAB Session

You can use only one version of CCS in a single MATLAB session. Embedded IDE Link does not support using multiple versions of CCS in a MATLAB session. To use another CCS version, exit MATLAB software and restart it. Then create your links to the new version of CCS.

MATLAB Hangs When Code Composer Studio Cannot Find a Board

In MATLAB software, when you create a `ticcs` object, the construction process for the object automatically starts CCS. If CCS cannot find a processor that is connected to your PC, you see a message from CCS like the following DSP Device Driver dialog box that indicates CCS could not initialize the processor.



Four options let you decide how to respond to the failure:

- **Abort** — Closes CCS and suspends control for about 30 seconds. If you used MATLAB software functions to open CCS, such as when you create a `ticcs` object, the system returns control to MATLAB command window after a considerable delay, and issues this warning:

```
??? Unable to establish connection with Code Composer Studio.
```

- **Ignore** — Starts CCS without connecting to any processor. In the CCS IDE you see a status message that says `EMULATOR DISCONNECTED` in the status area of the IDE. If you used MATLAB to start CCS, you get control immediately and Embedded IDE Link creates the `ticcs` object. Because

CCS is not connected to a processor, you cannot use the object to perform processor operations from MATLAB, such as loading or running programs.

- **Retry** — CCS tries again to initialize the processor. If CCS continues not to find your hardware processor, the same DSP Device Driver dialog box reappears. This process continues until either CCS finds the processor or you choose one of the other options to respond to the warning.

One more option, **Diagnostic**, lets you enter diagnostic mode if it is enabled. Usually, **Diagnostic** is not available for you to use.

Using Mapped Drives

Limitations in Code Composer Studio do not allow you to load programs after you set your CCS working directory to a read-only mapped drive. When you set the CCS working directory to a mapped drive for which you do not have write permissions, you cannot load programs from any location. Load operations fail with an Application Error dialog box.

The following combination of commands does not work:

- 1 `cd(cc,'mapped_drive_letter')` % Change CCS working directory to read-only mapped drive.
- 2 `load(cc,'program_file')` % Loading any program fails.

Uninstalling Code Composer Studio 3.3 Prevents Embedded IDE Link From Connecting

Description On a machine where CCS V3.3 and CCS V3.1 are installed, uninstalling V3.3 makes V3.1 unusable from MATLAB. This is because the CCS V3.3 uninstaller leaves stale registry entries in the Windows Registry that prevent MATLAB from connecting to CCS V3.1.

Texas Instruments has documented this uninstall problem and the solution on their Web site at <http://www-k.ext.ti.com/SRV5/CGI-BIN/WEBCGI.EXE/,?St=76,E=0000000000008373418>

Updated information on this issue may also be available from the Bug Reports section of www.mathworks.com at <http://www.mathworks.com/support/bugreports/379676>

A

- activate 7-2
- add 7-4
- address 7-6
- address, read 7-108
- animate 7-9
- apiversion 2-48
- Archive_library 3-60
- asynchronous scheduling 3-5

B

- block limitations using model reference 3-61
- boardnum 2-49
- boards, selecting 3-3
- build 7-10
- build configuration
 - compiler options, default 3-47
 - custom 3-47
 - default 3-47
- build configuration, new 7-89

C

- C280x/C28x3x hardware interrupt block 9-2
- C280x/C28x3x Hardware Interrupt block 9-2
- c281x hardware interrupt block 9-8
- C6000 model reference 3-59
- C6711 DSK
 - TLC debugging options 3-39
- CCS IDE objects
 - tutorial about using 2-2
- CCS status 7-70
- ccsappexe 2-49
- ccsboardinfo 7-14
- cd 7-20
- channel, open 7-92
- close 7-22
- Code Composer Studio
 - MATLAB API 1-3

- code profiling 7-95
- configuration parameters
 - pane 10-4
 - buildAction 10-21
 - Compiler options string: 10-16
 - configPILBlockAction 10-27
 - DiagnosticActions 10-33
 - Export IDE link handle to base workspace: 10-5
 - gui item name 10-12
 - IDE link handle name: 10-7
 - ideObjBuildTimeout 10-29
 - ideObjTimeout 10-31
 - Interrupt overrun notification function: 10-26
 - Linker options string: 10-18
 - overrunNotificationMethod 10-24
 - Profile real-time execution 10-8
 - profileBy 10-10
 - projectOptions 10-14
 - System stack size (MAUs): 10-20
- configure 7-25
- configure the software timer 9-47
- CPU clock speed 9-47
- create custom target function library 3-58
- current CPU clock speed 9-47
- custom build configuration 3-47
- custom compiler options 3-47
- custom data types 2-54
- custom source code 3-48
- custom type definitions 2-54

D

- Data Type Manager 2-54
- data types
 - managing 2-54
- datatypemanager 7-27
- debug point, insert 7-56
- default build configuration 3-47

- default compiler options 3-47
- dir 7-42
- disable 7-43
- discrete solver 3-33
- display 7-45
- DSP/BIOS
 - adding to generated code 3-42

E

- Embedded IDE Link™
 - code generation options 3-42
 - listing link functions 2-41
 - run-time options 3-42
- enable 7-47
- execute program 7-135
- execution in timer-based models 3-10
- execution profiling
 - subsystem 4-12
 - task 4-10
- export filters to CCS IDE from FDATool 5-1
 - select the export data type 5-6
 - set the Export mode option 5-5

F

- FDATool. *See* export filters to CCS IDE from FDATool
- file, new 7-89
- file, remove 7-131
- file, save 7-139
- fixed-step solver 3-33
- flush 7-49
- functions
 - overloading 2-45

G

- GEL file, load 7-86
- generate optimized code 3-42
- generate_code_only option 3-42

- get symbol table 7-141

H

- halt 7-51
- Hardware Interrupt block 9-14

I

- Idle Task block 9-17
- import filter coefficients from FDATool.. *See* FDATool
- info 7-53
- insert 7-56
- intrinsic. *See* target function library
- isenabled 7-60
- isreadable 7-62
- isrtdxcapable 7-67
- isrunning 7-68
- issues, using PIL 4-7
- isvisible 7-70
- iswritable 7-72

L

- link properties
 - about 2-46 2-48
 - apiversion 2-48
 - boardnum 2-49
 - ccsappexe 2-49
 - numchannels 2-49
 - page 2-50
 - procnum 2-50
 - quick reference table 2-46
 - rtdx 2-51
 - rtdxchannel 2-52
 - timeout 2-52
 - version 2-52
- link properties, details about 2-48
- links
 - closing CCS IDE 2-18

- closing RTDX 2-38
- communications for RTDX 2-29
- creating links for RTDX 2-26
- details 2-48
- introducing the tutorial for using links for RTDX 2-21
- loading files into CCS IDE 2-10
- quick reference 2-46
- running applications using RTDX 2-31
- tutorial about using links for RTDX 2-20
- working with your processor 2-12

- list 7-76
- list object 7-76
- list variable 7-76
- load 7-86

M

- manage data types 7-27
- managing data types 2-54
- matrix, read from RTDX 7-114
- Memory Allocate block 9-20
- Memory Copy block 9-26
- memory, write 7-153
- model execution 3-5
- model reference 3-59
 - about 3-59
 - Archive_library 3-60
 - block limitations 3-61
 - modelreferencecompliant flag 3-62
 - setting build action 3-60
 - Target Preferences blocks 3-61
 - using 3-60
- model schedulers 3-5
- modelreferencecompliant flag 3-62
- msgcount 7-88

N

- new

- build configuration 7-89
 - file 7-89
 - project 7-89
- numchannels 2-49

O

- object
 - ticcs 2-42
- object, read 7-108
- objects
 - creating objects for CCS IDE 2-8
 - introducing the objects for CCS IDE tutorial 2-2
 - selecting processors for CCS IDE 2-6
 - tutorial about using Automation Interface for CCS IDE 2-2
- open channel 7-92
- optimization, processor specific 3-42
- overloading 2-45

P

- page 2-50
- PIL block 4-4
- PIL cosimulation
 - overview 4-3
- PIL issues 4-7
- process, halt 7-51
- processor
 - general code generation options 3-40
- processor configuration options
 - build action 3-42
 - generate code only 3-38
 - overrun action 3-44
 - system target file 3-37
- processor function library. *See* target function library
- processor information, get 7-53
- processor specific optimization 3-42

- processor status 7-68
- processor, reset 7-132
- processor, write 7-153
- procnum 2-50
- profile 7-95
- profiling code 7-95
- profiling execution
 - by subsystem 4-12
 - by task 4-10
- program counter, restore 7-133
- program file, load 7-86
- program file, reload 7-129
- program, run 7-135
- project generation
 - selecting the board 3-3
- project, new 7-89
- project, save 7-139
- properties
 - link properties 2-46

R

- read
 - address 7-108
 - object 7-108
- read register 7-121
- readmat 7-114
- readmsg 7-117
- Real-Time Workshop solver options 3-33
- Real-Time Workshop build options
 - generate_code_only 3-42
- regread 7-121
- regwrite 7-125
- reload 7-129
- remove 7-131
- remove file 7-131
- replacing generated code 3-48
- replacing linker directives 3-48
- reset 7-132
- restart 7-133

- restore program counter 7-133
- rt dx 2-51
- RTDX
 - isenabled 7-60
 - isrtdxcapable 7-67
 - message count 7-88
 - open channel 7-92
 - read message 7-117
 - readmat 7-114
 - writemsg 7-158
- RTDX channel, flush 7-49
- RTDX links
 - tutorial about using 2-20
- RTDX message count 7-88
- RTDX, disable 7-43
- RTDX, enable 7-47
- rt dxchannel 2-52
- run 7-135

S

- save 7-139
- selecting boards 3-3
- set stack size 3-45
- set visibility 7-151
- solver option settings 3-33
- source code replacement 3-48
- stack size, set stack size 3-45
- stop process 7-51
- symbol 7-141
- symbol table, getting symbols 7-141
- synchronous scheduling 3-10

T

- target function library
 - assessing execution time after selecting a library 3-55
 - create a custom library 3-58
 - optimization 3-52

- seeing the library changes in your generated code 3-56
 - selecting the library to use 3-54
 - use in the build process 3-53
 - using with link software 3-52
 - viewing library tables 3-58
 - when to use 3-54
- Target Preferences blocks in referenced models 3-61
- Target Preferences/Custom Board block 9-45
- TFL. *See* target function library
- ticcs 2-42 7-143
- timeout 2-52
- timer, configure 9-47
- timer-based models, execution 3-10
- timer-based scheduler 3-10
- timing 3-5
- tutorials
 - links for RTDX 2-20
 - objects for CCS 2-2
- typedefs 2-56
 - about 2-54
 - adding 2-56
 - managing 2-56
 - removing 2-56
- V**
- version 2-52
- view CCS 7-70
- viewing target function libraries 3-58
- visibility, setting 7-151
- visible 7-151
- W**
- write 7-153
- write register 7-125
- writemsg 7-158